CreateProcess QuickInfo Overview Group

The **CreateProcess** function creates a new process and its primary thread. The new process executes the specified executable file.

```
BOOL CreateProcess(
  LPCTSTR  lpApplicationName,                 // pointer to name of executable module
  LPTSTR  lpCommandLine,                      // pointer to command line string
  LPSECURITY_ATTRIBUTES  lpProcessAttributes, // pointer to process security attributes
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,  // pointer to thread security attributes
  BOOL  bInheritHandles,                      // handle inheritance flag
  DWORD  dwCreationFlags,                     // creation flags
  LPVOID  lpEnvironment,                      // pointer to new environment block
  LPCTSTR  lpCurrentDirectory,                // pointer to current directory name
  LPSTARTUPINFO  lpStartupInfo,               // pointer to STARTUPINFO
  LPPROCESS_INFORMATION  lpProcessInformation // pointer to PROCESS_INFORMATION
);
```

Parameters

lpApplicationName

Pointer to a null-terminated string that specifies the module to execute.

The string can specify the full path and filename of the module to execute.

The string can specify a partial name. In that case, the function uses the current drive and current directory to complete the specification.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white space-delimited token in the *lpCommandLine* string.

The specified module can be a Win32-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.

Windows NT : 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. A 16-bit application is one that executes as a VDM or WOW process.

lpCommandLine

Pointer to a null-terminated string that specifies the command line to execute.

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. C runtime processes can use the **argc** and **argv** arguments.

If *lpApplicationName* is NULL, the first white space-delimited token of the command line specifies the module name. If the filename does not contain an extension, .EXE is assumed. If the filename ends in a period (.) with no extension, or the filename contains a path, .EXE is not appended. If the filename does not contain a directory path, Windows 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. **Windows 95:** The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

   Windows NT: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.
4. **Windows NT only:** The 16-bit Windows system directory. There is no Win32 function that

4. **Windows NT only:** The 16-bit Windows system directory. There is no Win32 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.

If the process to be created is an MS-DOS - based or Windows-based application, *lpCommandLine* should be a full command line in which the first element is the application name. Because this also works well for Win32-based applications, it is the most robust way to set *lpCommandLine*.

lpProcessAttributes

Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the created process.

If *lpProcessAttributes* is NULL, the process is created with a default security descriptor, and the resulting handle is not inherited.

lpThreadAttributes

Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the primary thread of the new process. If *lpThreadAttributes* is NULL, the process is created with a default security descriptor, and the resulting handle is not inherited.

bInheritHandles

Indicates whether the new process inherits handles from the calling process. If TRUE, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

dwCreationFlags

Specifies additional flags that control the priority class and the creation of the process. The following creation flags can be specified in any combination, except as noted:

Value	Meaning
CREATE_DEFAULT_ERROR_MODE	The new process does not inherit the error mode of the calling process. Instead, **CreateProcess** gives the new process the current default error mode. An application sets the current default error mode by calling **SetErrorMode**. This flag is particularly useful for multi-threaded shell applications that run with hard errors disabled. The default behavior for **CreateProcess** is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.
CREATE_NEW_CONSOLE	The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with the DETACHED_PROCESS flag.
CREATE_NEW_PROCESS_GROUP	The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process ID of the new process group is the same as the process ID, which is returned in the *lpProcessInformation* parameter. Process groups are used by the **GenerateConsoleCtrlEvent** function to enable sending a CTRL+C or CTRL+BREAK signal to a group of console processes.
CREATE_SEPARATE_WOW_VDM	This flag is only valid only launching a 16-bit Windows program. If set, the new process is run in a private Virtual

program. If set, the new process is run in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows programs are run in a single, shared VDM. The advantage of running separately is that a crash only kills the single VDM; any other programs running in distinct VDMs continue to function normally. Also, 16-bit Windows applications which are run in separate VDMs have separate input queues. That means that if one application hangs momentarily, applications in separate VDMs continue to receive input.

CREATE_SHARED_WOW_VDM

Windows NT: The flag is valid only when launching a 16-bit Windows program. If the DefaultSeparateVDM switch in the Windows section of WIN.INI is TRUE, this flag causes the **CreateProcess** function to override the switch and run the new process in the shared Virtual DOS Machine.

CREATE_SUSPENDED

The primary thread of the new process is created in a suspended state, and does not run until the **ResumeThread** function is called.

CREATE_UNICODE_ENVIRONMENT

If set, the environment block pointed to by *lpEnvironment* uses Unicode characters. If clear, the environment block uses ANSI characters.

DEBUG_PROCESS

If this flag is set, the calling process is treated as a debugger, and the new process is a process being debugged. The system notifies the debugger of all debug events that occur in the process being debugged.

If you create a process with this flag set, only the calling thread (the thread that called **CreateProcess**) can call the **WaitForDebugEvent** function.

DEBUG_ONLY_THIS_PROCESS

If not set and the calling process is being debugged, the new process becomes another process being debugged by the calling process's debugger. If the calling process is not a process being debugged, no debugging-related actions occur.

DETACHED_PROCESS

For console processes, the new process does not have access to the console of the parent process. The new process can call the **AllocConsole** function at a later time to create a new console. This flag cannot be used with the CREATE_NEW_CONSOLE flag.

The *dwCreationFlags* parameter also controls the new process's priority class, which is used in determining the scheduling priorities of the process's threads. If none of the following 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**. In this case the default priority class of the child process is **IDLE_PRIORITY_CLASS**. One of the following flags can be specified:

Priority	**Meaning**
HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a

	correctly. The threads of a high-priority class process preempt the threads of normal-priority or idle-priority class processes. An example is Windows 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	Indicates a 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	Indicates a normal process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Indicates a 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.

lpEnvironment

Points 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 form:

```
name=value
```

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

If an application provides an environment block, rather than passing NULL for this parameter, the current directory information of the system drives is not automatically propagated to the new process. For a discussion of this situation and how to handle it, see the following Remarks section.

An environment block can contain Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, the *dwCreationFlags* field's CREATE_UNICODE_ENVIRONMENT flag will be set. If the block contains ANSI characters, that flag will be clear.

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.

lpCurrentDirectory

Points to a null-terminated string that specifies the current drive and directory for the child process. The string must be a full path and filename that includes a drive letter. If this parameter is NULL,

The string must be a full path and filename that includes a drive letter. If this parameter is NULL, the new process is created with the same current drive and directory as the calling process. This option is provided primarily for shells that need to start an application and specify its initial drive and working directory.

lpStartupInfo

Points to a **STARTUPINFO** structure that specifies how the main window for the new process should appear.

lpProcessInformation

Points to a **PROCESS_INFORMATION** structure that receives identification information about the new process.

Return Value

If the function succeeds, the return value is TRUE.

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

Remarks

The **CreateProcess** function is used to run a new program. The **WinExec** and **LoadModule** functions are still available, but they are implemented as calls to **CreateProcess**.

In addition to creating a process, **CreateProcess** also creates a thread object. The thread is created with an initial stack whose size is described in the image header of the specified program's executable file. The thread begins execution at the image's entry point.

The new process and the new thread handles are created with full access rights. For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle of that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process is not able to use the handle to gain access to the thread.

The process is assigned a 32-bit process identifier. The ID 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 32-bit thread identifier. The ID 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.

When specifying an application name in the *lpApplicationName* or *lpCommandLine* strings, it doesn't matter whether the application name includes the filename extension, with one exception: an MS-DOS – based or Windows-based application whose filename extension is .COM must include the .COM extension.

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 notifies all dynamic-link libraries (DLLs) attached to the process of the approaching termination. 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).

ExitProcess, **ExitThread**, **CreateThread**, **CreateRemoteThread**, 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 the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not

- 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.
- The **ExitProcess** function does not return until no threads are in their DLL initialization or detach routines.

The created process remains in the system until all threads within the process have terminated and all handles to the process and any of its threads have been closed through calls to **CloseHandle**. The handles for both the process and the main thread must be closed through calls to **CloseHandle**. If these handles are not needed, it is best to close them immediately after the process is created.

When the last thread in a process terminates, the following events occur:

- All objects opened by the process are implicitly closed.
- The process's termination status (which is returned by **GetExitCodeProcess**) changes from its initial value of STILL_ACTIVE to the termination status of the last thread to terminate.
- The thread object of the main thread is set to the signaled state, satisfying any threads that were waiting on the object.
- The process object is set to the signaled state, satisfying any threads that were waiting on the object.

If the current directory on drive C is \MSVC\MFC, there is an environment variable called =C: whose value is C:\MSVC\MFC. As noted in the previous description of *lpEnvironment*, such current directory information for a system's drives does not automatically propagate to a new process when the **CreateProcess** function's *lpEnvironment* parameter is non-NULL. An application must manually pass the current directory information to the new process. To do so, the application must explicitly create the =X environment variable strings, get them into alphabetical order (because Windows NT and Windows 95 use a sorted environment), and then put them into the environment block specified by *lpEnvironment*. Typically, they will go at the front of the environment block, due to the previously mentioned environment block sorting.

One way to obtain the current directory variable for a drive X is to 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.

The handle returned by the **CreateProcess** function has PROCESS_ALL_ACCESS access to the process object.

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 good 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.

The current directory specified by the *lpcurrentDirectory* parameter is the current directory for the child process. The current directory specified in item 2 under the *lpCommandLine* parameter is the current directory for the parent process.

See Also

AllocConsole, **CloseHandle**, **CreateRemoteThread**, **CreateThread**, **ExitProcess**, **ExitThread**, **GenerateConsoleCtrlEvent**, **GetCommandLine**, **GetEnvironmentStrings**, **GetExitCodeProcess**, **GetFullPathName**, **GetStartupInfo**, **GetSystemDirectory**, **GetWindowsDirectory**, **LoadModule**, **OpenProcess**, **PROCESS_INFORMATION**, **ResumeThread**, **SECURITY_ATTRIBUTES**, **SetConsoleCtrlHandler**, **SetErrorMode**, **STARTUPINFO**, **TerminateProcess**, **WaitForInputIdle**, **WaitForDebugEvent**, **WinExec**