

CreateMutexA function (synchapi.h)

05/03/2020 • 5 minutes to read

In this article

Syntax

Parameters

Return value

Remarks


Requirements

See also

Creates or opens a named or unnamed mutex object.

To specify an access mask for the object, use the `CreateMutexEx` function.

Syntax

| C++ |  Copy |
|--|--|
| <pre>HANDLE CreateMutexA(LPSECURITY_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCSTR lpName);</pre> | |

Parameters

`lpMutexAttributes`

A pointer to a `SECURITY_ATTRIBUTES` structure. If this parameter is **NULL**, the handle cannot be inherited by child processes.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new mutex. If *lpMutexAttributes* is **NULL**, the mutex gets a default security descriptor. The ACLs in the default security descriptor for a mutex come from the primary or impersonation token of the creator. For more information, see [Synchronization Object Security and Access Rights](#).

`bInitialOwner`

If this value is **TRUE** and the caller created the mutex, the calling thread obtains initial ownership of the mutex object. Otherwise, the calling thread does not obtain ownership of the mutex. To determine if the caller created the mutex, see the Return Values section.

`lpName`

The name of the mutex object. The name is limited to **MAX_PATH** characters. Name comparison is case sensitive.

If *lpName* matches the name of an existing named mutex object, this function requests the **MUTEX_ALL_ACCESS** access right. In this case, the *bInitialOwner* parameter is ignored because it has already been set by the creating process. If the *lpMutexAttributes* parameter is not **NULL**, it determines whether the handle can be inherited, but its security-descriptor member is ignored.

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

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

The name can have a "Global" or "Local" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Namespaces. Fast user switching is implemented using Terminal Services sessions. Kernel object names must follow the guidelines outlined for Terminal Services so that applications can support multiple users.

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

Return value

If the function succeeds, the return value is a handle to the newly created mutex object.

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

If the mutex is a named mutex and the object existed before this function call, the return value is a handle to the existing object, and the `GetLastError` function returns

ERROR_ALREADY_EXISTS.

Remarks

The handle returned by **CreateMutex** has the **MUTEX_ALL_ACCESS** access right; it can be used in any function that requires a handle to a mutex object, provided that the caller has been granted access. If a mutex is created from a service or a thread that is impersonating a different user, you can either apply a security descriptor to the mutex when you create it, or change the default security descriptor for the creating process by changing its default DACL. For more information, see [Synchronization Object Security and Access Rights](#).

If you are using a named mutex to limit your application to a single instance, a malicious user can create this mutex before you do and prevent your application from starting. To prevent this situation, create a randomly named mutex and store the name so that it can only be obtained by an authorized user. Alternatively, you can use a file for this purpose. To limit your application to one instance per user, create a locked file in the user's profile directory.

Any thread of the calling process can specify the mutex-object handle in a call to one of the wait functions. The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution.

The state of a mutex object is signaled when it is not owned by any thread. The creating thread can use the *InitialOwner* flag to request immediate ownership of the mutex. Otherwise, a thread must use one of the wait functions to request ownership. When the mutex's state is signaled, one waiting thread is granted ownership, the mutex's state changes to nonsignaled, and the wait function returns. Only one thread can own a mutex at any given time. The owning thread uses the **ReleaseMutex** function to release its ownership.

The thread that owns a mutex can specify the same mutex in repeated wait function calls without blocking its execution. Typically, you would not wait repeatedly for the same mutex, but this mechanism prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call **ReleaseMutex** once for each time that the mutex satisfied a wait.

Two or more processes can call **CreateMutex** to create the same named mutex. The first process actually creates the mutex, and subsequent processes with sufficient access rights

simply open a handle to the existing mutex. This enables multiple processes to get handles of the same mutex, while relieving the user of the responsibility of ensuring that the creating process is started first. When using this technique, you should set the *InitialOwner* flag to **FALSE**; otherwise, it can be difficult to be certain which process has initial ownership.

Multiple processes can have handles of the same mutex object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

- A child process created by the `CreateProcess` function can inherit a handle to a mutex object if the *lpMutexAttributes* parameter of **CreateMutex** enabled inheritance. This mechanism works for both named and unnamed mutexes.
- A process can specify the handle to a mutex object in a call to the `DuplicateHandle` function to create a duplicate handle that can be used by another process. This mechanism works for both named and unnamed mutexes.
- A process can specify a named mutex in a call to `[OpenMutex](./nf-synchapi-openmutexw.md)` or **CreateMutex** to retrieve a handle to the mutex object.

Use the `CloseHandle` function to close the handle. The system closes the handle automatically when the process terminates. The mutex object is destroyed when its last handle has been closed.

Examples

See [Using Mutex Objects](#) for an example of **CreateMutex**.

ⓘ Note

The `synchapi.h` header defines `CreateMutex` 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 UWP apps] |
| Minimum supported server | Windows Server 2003 [desktop apps UWP apps] |
| Target Platform | Windows |
| Header | synchapi.h (include Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2, Windows.h) |
| Library | Kernel32.lib |
| DLL | Kernel32.dll |

See also

[CloseHandle](#)

[CreateMutexEx](#)

[CreateProcess](#)

[DuplicateHandle](#)

[Mutex Objects](#)

[Object Names](#)

[OpenMutex](#)

[ReleaseMutex](#)

[SECURITY_ATTRIBUTES](#)

[Synchronization Functions](#)

Is this page helpful?

 Yes  No

Recommended content

ReleaseMutex function (synchapi.h) - Win32 apps

Releases ownership of the specified mutex object.

OpenFileMappingA function (winbase.h) - Win32 apps

Opens a named file mapping object.

CreateThread function (processthreadsapi.h) - Win32 apps

Creates a thread to execute within the virtual address space of the calling process.

TerminateThread function (processthreadsapi.h) - Win32 apps

Terminates a thread.

Show more 