Synchronization

There are a variety of ways to coordinate multiple threads of execution. The functions described in this overview provide mechanisms that threads can use to synchronize access to a resource.

- What's New in Synchronization
- About Synchronization
- Using Synchronization
- Synchronization Reference

About Synchronization

To synchronize access to a resource, use one of the synchronization objects in one of the wait functions. The state of a synchronization object is either *signaled* or *nonsignaled*. The wait functions allow a thread to block its own execution until a specified nonsignaled object is set to the signaled state. For more information, see Interprocess Synchronization.

The following are other synchronization mechanisms:

- overlapped input and output
- asynchronous procedure calls
- critical section objects
- condition variables
- slim reader/writer locks
- one-time initialization
- interlocked variable access
- interlocked singly linked lists
- timer queues
- the **MemoryBarrier** macro

For additional information on synchronization, see Synchronization and Multiprocessor Issues.

Wait Functions

Wait functions allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state until the conditions of the wait criteria have been met or the specified time-out interval elapses.

- Single-object Wait Functions
- Multiple-object Wait Functions
- Alertable Wait Functions
- Registered Wait Functions
- Waiting on an Address
- Wait Functions and Time-out Intervals
- Wait Functions and Synchronization Objects
- Wait Functions and Creating Windows

Single-object Wait Functions

The **SignalObjectAndWait**, **WaitForSingleObject**, and **WaitForSingleObjectEx**functions require a handle to one synchronization object. These functions return when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses. The time-out interval can be set to **INFINITE** to specify that the wait will not time out.

The **SignalObjectAndWait** function enables the calling thread to atomically set the state of an object to signaled and wait for the state of another object to be set to signaled.

Multiple-object Wait Functions

The **WaitForMultipleObjects**, **WaitForMultipleObjectsEx**, **MsgWaitForMultipleObjects**, and **MsgWaitForMultipleObjectsEx** functions enable the calling thread to specify an array containing one or more synchronization object handles. These functions return when one of the following occurs:

- The state of any one of the specified objects is set to signaled or the states of all objects have been set to signaled. You control whether one or all of the states will be used in the function call.

- The time-out interval elapses. The time-out interval can be set to **INFINITE** to specify that the wait will not time out.

The **MsgWaitForMultipleObjects** and **MsgWaitForMultipleObjectsEx** function allow you to specify input event objects in the object handle array. This is done when you specify the type of input to wait for in the thread's input queue. For example, a thread could use **MsgWaitForMultipleObjects** to block its execution until the state of a specified object has been set to signaled and there is mouse input available in the thread's input queue. The thread can use the **GetMessage** or **PeekMessage** function to retrieve the input.

When waiting for the states of all objects to be set to signaled, these multiple-object functions do not modify the states of the specified objects until the states of all objects have been set signaled. For example, the state of a mutex object can be signaled, but the calling thread does not get ownership until the states of the other objects specified in the array have also been set to signaled. In the meantime, some other thread may get ownership of the mutex object, thereby setting its state to nonsignaled.

When waiting for the state of a single object to be set to signaled, these multiple-object functions check the handles in the array in order starting with index 0, until one of the objects is signaled. If multiple objects become signaled, the function returns the index of the first handle in the array whose object was signaled.

Alertable Wait Functions

The **MsgWaitForMultipleObjectsEx**, **SignalObjectAndWait**, **WaitForMultipleObjectsEx**, and **WaitForSingleObjectEx** functions differ from the other wait functions in that they can optionally perform an *alertable wait operation*. In an alertable wait operation, the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread. For more information about alertable wait operations and I/O completion routines, see Synchronization and Overlapped Input and Output. For more information about APCs, see Asynchronous Procedure Calls.

Registered Wait Functions

The **RegisterWaitForSingleObject** function differs from the other wait functions in that the wait operation is performed by a thread from the thread pool. When the specified conditions are met, the callback function is executed by a worker thread from the thread pool.

By default, a registered wait operation is a multiple-wait operation. The system resets the timer every time the event is signaled (or the time-out interval elapses) until you call

the **UnregisterWaitEx** function to cancel the operation. To specify that a wait operation should be executed only once, set the *dwFlags* parameter of **RegisterWaitForSingleObject** to **WT_EXECUTEONLYONCE**.

If the thread calls functions that use APCs, set the *dwFlags* parameter of **RegisterWaitForSingleObject** to **WT_EXECUTEINPERSISTENTTHREAD**.

Waiting on an Address

A thread can use the **WaitOnAddress** function to wait for the value of a target address to change from some undesired value to any other value. This enables threads to wait for a value to change without having to spin or handle the synchronization problems that can arise when the thread captures an undesired value but the value changes before the thread can wait.

WaitOnAddress returns when code that modifies the target value signals the change by calling **WakeByAddressSingle** to wake a single waiting thread or **WakeByAddressAll** to wake all waiting threads. If a time-out interval is specified with **WaitOnAddress** and no thread calls a wake function, the function returns when the time-out interval elapses. If no time-out interval is specified, the thread waits indefinitely.

Wait Functions and Time-out Intervals

The accuracy of the specified time-out interval depends on the resolution of the system clock. The system clock "ticks" at a constant rate. If the time-out interval is less than the resolution of the system clock, the wait may time out in less than the specified length of time. If the time-out interval is greater than one tick but less than two, the wait can be anywhere between one and two ticks, and so on.

To increase the accuracy of the time-out interval for the wait functions, call the **timeGetDevCaps** function to determine the supported minimum timer resolution and the **timeBeginPeriod** function to set the timer resolution to its minimum. Use caution when calling **timeBeginPeriod**, as frequent calls can significantly affect the system clock, system power usage, and the scheduler. If you call **timeBeginPeriod**, call it one time early in the application and be sure to call the **timeEndPeriod**function at the very end of the application.

Wait Functions and Synchronization Objects

The wait functions can modify the states of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the

function to return. Wait functions can modify the states of synchronization objects as follows:

- The count of a semaphore object decreases by one, and the state of the semaphore is set to nonsignaled if its count is zero.
- The states of mutex, auto-reset event, and change-notification objects are set to nonsignaled.
- The state of a synchronization timer is set to nonsignaled.
- The states of manual-reset event, manual-reset timer, process, thread, and console input objects are not affected by a wait function.

Wait Functions and Creating Windows

You have to be careful when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses a wait function with no time-out interval, the system will deadlock. Two examples of code that indirectly creates windows are DDE and the **CoInitialize** function. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than the other wait functions.

Synchronization Objects

A *synchronization object* is an object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads. More than one process can have a handle to the same synchronization object, making interprocess synchronization possible.

The following object types are provided exclusively for synchronization.

Type	Description
Event	Notifies one or more waiting threads that an event has occurred. For more information, see Event Objects.
Mutex	Can be owned by only one thread at a time, enabling threads to coordinate mutually exclusive access to a shared resource. For more information, see Mutex Objects.
Semaphore	Maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource. For more information, see Semaphore Objects.
Waitable timer	Notifies one or more waiting threads that a specified time has arrived. For more information, see Waitable Timer Objects.

Though available for other uses, the following objects can also be used for synchronization.

Object	Description
Change notification	Created by the **FindFirstChangeNotification** function, its state is set to signaled when a specified type of change occurs within a specified directory or directory tree. For more information, see Obtaining Directory Change Notifications.
Console input	Created when a console is created. The handle to console input is returned by the **CreateFile** function when CONIN$ is specified, or by the **GetStdHandle** function. Its state is set to signaled when there is unread input in the console's input buffer, and set to

Object	Description
	nonsignaled when the input buffer is empty. For more information about consoles, see Character-Mode Applications
Job	Created by calling the **CreateJobObject** function. The state of a job object is set to signaled when all its processes are terminated because the specified end-of-job time limit has been exceeded. For more information about job objects, see Job Objects.
Memory resource notification	Created by the **CreateMemoryResourceNotification** function. Its state is set to signaled when a specified type of change occurs within physical memory. For more information about memory, see Memory Management.
Process	Created by calling the **CreateProcess** function. Its state is set to nonsignaled while the process is running, and set to signaled when the process terminates. For more information about processes, see Processes and Threads.
Thread	Created when a new thread is created by calling the **CreateProcess**, **CreateThread**, or **CreateRemoteThread** function. Its state is set to nonsignaled while the thread is running, and set to signaled when the thread terminates. For more information about threads, see Processes and Threads.

In some circumstances, you can also use a file, named pipe, or communications device as a synchronization object; however, their use for this purpose is discouraged. Instead, use asynchronous I/O and wait on the event object set in the **OVERLAPPED** structure. It is safer to use the event object because of the confusion that can occur when multiple simultaneous overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.

For additional information about I/O operations on files, named pipes, or communications, see Synchronization and Overlapped Input and Output.

Event Objects

An *event object* is a synchronization object whose state can be explicitly set to signaled by use of the **SetEvent** function. Following are the two types of event object.

Object	Description
Manual-reset event	An event object whose state remains signaled until it is explicitly reset to nonsignaled by the **ResetEvent** function. While it is signaled, any number of waiting threads, or threads that subsequently specify the same event object in one of the wait functions, can be released.
Auto-reset event	An event object whose state remains signaled until a single waiting thread is released, at which time the system automatically sets the state to nonsignaled. If no threads are waiting, the event object's state remains signaled. If more than one thread is waiting, a waiting thread is selected. Do not assume a first-in, first-out (FIFO) order. External events such as kernel-mode APCs can change the wait order.

The event object is useful in sending a signal to a thread indicating that a particular event has occurred. For example, in overlapped input and output, the system sets a specified event object to the signaled state when the overlapped operation has been completed. A single thread can specify different event objects in several simultaneous overlapped operations, then use one of the multiple-object wait functions to wait for the state of any one of the event objects to be signaled.

A thread uses the **CreateEvent** or **CreateEventEx** function to create an event object. The creating thread specifies the initial state of the object and whether it is a manual-reset or auto-reset event object. The creating thread can also specify a name for the event object. Threads in other processes can open a handle to an existing event object by specifying its name in a call to the **OpenEvent** function. For additional information about names for mutex, event, semaphore, and timer objects, see Interprocess Synchronization.

Related topics

Using Event Objects

Mutex Objects

A *mutex object* is a synchronization object whose state is set to signaled when it is not owned by any thread, and nonsignaled when it is owned. Only one thread at a time can own a mutex object, whose name comes from the fact that it is useful in coordinating mutually exclusive access to a shared resource. For example, to prevent two threads from writing to shared memory at the same time, each thread waits for ownership of a mutex object before executing the code that accesses the memory. After writing to the shared memory, the thread releases the mutex object.

A thread uses the **CreateMutex** or **CreateMutexEx** function to create a mutex object. The creating thread can request immediate ownership of the mutex object and can also specify a name for the mutex object. It can also create an unnamed mutex. For additional information about names for mutex, event, semaphore, and timer objects, see Interprocess Synchronization.

Threads in other processes can open a handle to an existing named mutex object by specifying its name in a call to the **OpenMutex** function. To pass a handle to an unnamed mutex to another process, use the **DuplicateHandle** function or parent-child handle inheritance.

Any thread with a handle to a mutex object can use one of the wait functions to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object using the **ReleaseMutex** function. The return value of the wait function indicates whether the function returned for some reason other than the state of the mutex being set to signaled.

If more than one thread is waiting on a mutex, a waiting thread is selected. Do not assume a first-in, first-out (FIFO) order. External events such as kernel-mode APCs can change the wait order.

After a thread obtains ownership of a mutex, it can specify the same mutex in repeated calls to the wait-functions without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. To release its ownership under such circumstances, the thread must call **ReleaseMutex** once for each time that the mutex satisfied the conditions of a wait function.

If a thread terminates without releasing its ownership of a mutex object, the mutex object is considered to be abandoned. A waiting thread can acquire ownership of an

abandoned mutex object, but the wait function will return **WAIT_ABANDONED** to indicate that the mutex object is abandoned. An abandoned mutex object indicates that an error has occurred and that any shared resource being protected by the mutex object is in an undefined state. If the thread proceeds as though the mutex object had not been abandoned, it is no longer considered abandoned after the thread releases its ownership. This restores normal behavior if a handle to the mutex object is subsequently specified in a wait function.

Note that critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process.

Related topics

Using Mutex Objects

Semaphore Objects

A *semaphore object* is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore. When the count reaches zero, no more threads can successfully wait for the semaphore object state to become signaled. The state of a semaphore is set to signaled when its count is greater than zero, and nonsignaled when its count is zero.

The semaphore object is useful in controlling a shared resource that can support a limited number of users. It acts as a gate that limits the number of threads sharing the resource to a specified maximum number. For example, an application might place a limit on the number of windows that it creates. It uses a semaphore with a maximum count equal to the window limit, decrementing the count whenever a window is created and incrementing it whenever a window is closed. The application specifies the semaphore object in call to one of the wait functionsbefore each window is created. When the count is zero—indicating that the window limit has been reached—the wait function blocks execution of the window-creation code.

A thread uses the **CreateSemaphore** or **CreateSemaphoreEx** function to create a semaphore object. The creating thread specifies the initial count and the maximum value of the count for the object. The initial count must be neither less than zero nor greater than the maximum value. The creating thread can also specify a name for the semaphore object. Threads in other processes can open a handle to an existing semaphore object by specifying its name in a call to the **OpenSemaphore** function. For additional information about names for mutex, event, semaphore, and timer objects, see Interprocess Synchronization.

If more than one thread is waiting on a semaphore, a waiting thread is selected. Do not assume a first-in, first-out (FIFO) order. External events such as kernel-mode APCs can change the wait order.

Each time one of the wait functions returns because the state of a semaphore was set to signaled, the count of the semaphore is decreased by one.
The **ReleaseSemaphore** function increases a semaphore's count by a specified amount. The count can never be less than zero or greater than the maximum value.

The initial count of a semaphore is typically set to the maximum value. The count is then decremented from that level as the protected resource is consumed. Alternatively, you can create a semaphore with an initial count of zero to block access to the protected

resource while the application is being initialized. After initialization, you can use **ReleaseSemaphore** to increment the count to the maximum value.

A thread that owns a mutex object can wait repeatedly for the same mutex object to become signaled without its execution becoming blocked. A thread that waits repeatedly for the same semaphore object, however, decrements the semaphore's count each time a wait operation is completed; the thread is blocked when the count gets to zero. Similarly, only the thread that owns a mutex can successfully call the **ReleaseMutex** function, though any thread can use **ReleaseSemaphore** to increase the count of a semaphore object.

A thread can decrement a semaphore's count more than once by repeatedly specifying the same semaphore object in calls to any of the wait functions. However, calling one of the multiple-object wait functions with an array that contains multiple handles of the same semaphore does not result in multiple decrements.

When you have finished using the semaphore object, call the **CloseHandle**function to close the handle. The semaphore object is destroyed when its last handle has been closed. Closing the handle does not affect the semaphore count; therefore, be sure to call **ReleaseSemaphore** before closing the handle or before the process terminates. Otherwise, pending wait operations will either time out or continue indefinitely, depending on whether a time-out value has been specified.

Related topics

Using Semaphore Objects

Waitable Timer Objects

A *waitable timer object* is a synchronization object whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer.

Object	Description
manual-reset timer	A timer whose state remains signaled until **SetWaitableTimer** is called to establish a new due time.
synchronization timer	A timer whose state remains signaled until a thread completes a wait operation on the timer object.
periodic timer	A timer that is reactivated each time the specified period expires, until the timer is reset or canceled. A periodic timer is either a periodic manual-reset timer or a periodic synchronization timer.

Note

When a timer is signaled, the processor must run to process the associated instructions. High-frequency periodic timers keep the processor continually busy, which prevents the system from remaining in a lower **power state** for any meaningful amount of time. This can have a negative impact on portable computer battery life and scenarios that depend on effective power management, such as large datacenters. For greater energy efficiency, consider using event-based notifications instead of time-based notifications in your application. If a timer is necessary, use a timer that is signaled once rather than a periodic timer, or set the interval to a value greater than one second.

A thread uses the **CreateWaitableTimer** or **CreateWaitableTimerEx** function to create a timer object. The creating thread specifies whether the timer is a manual-reset timer or a synchronization timer. The creating thread can specify a name for the timer object. Threads in other processes can open a handle to an existing timer by specifying its

name in a call to the **OpenWaitableTimer** function. Any thread with a handle to a timer object can use one of the wait functions to wait for the timer state to be set to signaled.

- The thread calls the **SetWaitableTimer** function to activate the timer. Note the use of the following parameters for **SetWaitableTimer**:
- Use the *lpDueTime* parameter to specify the time at which the timer is to be set to the signaled state. When a manual-reset timer is set to the signaled state, it remains in this state until **SetWaitableTimer** establishes a new due time. When a synchronization timer is set to the signaled state, it remains in this state until a thread completes a wait operation on the timer object.
- Use the *lPeriod* parameter of the **SetWaitableTimer** function to specify the timer period. If the period is not zero, the timer is a periodic timer; it is reactivated each time the period expires, until the timer is reset or canceled. If the period is zero, the timer is not a periodic timer; it is signaled once and then deactivated.

A thread can use the **CancelWaitableTimer** function to set the timer to the inactive state. To reset the timer, call **SetWaitableTimer**. When you are finished with the timer object, call **CloseHandle** to close the handle to the timer object.

The behavior of a waitable timer can be summarized as follows:

- When a timer is set, it is canceled if it was already active, the state of the timer is nonsignaled, and the timer is placed in the kernel timer queue.
- When a timer expires, the timer is set to the signaled state. If the timer has a completion routine, it is queued to the thread that set the timer. The completion routine remains in the asynchronous procedure call (APC) queue of the thread until the thread enters an alertable wait state. At that time, the APC is dispatched and the completion routine is called. If the timer is periodic, it is placed back in the kernel timer queue.
- When a timer is canceled, it is removed from the kernel timer queue if it was pending. If the timer had expired and there is still an APC queued to the thread that set the timer, the APC is removed from the thread's APC queue. The signaled state of the timer is not affected.

Related topics

Asynchronous Procedure Calls

Using Waitable Timer Objects

Synchronization Object Security and Access Rights

The Windows security model enables you to control access to event, mutex, semaphore, and waitable timer objects. Timer queues, interlocked variables, and critical section objects are not securable. For more information, see Access-Control Model.

You can specify a security descriptor for an interprocess synchronization object when you call the **CreateEvent**, **CreateMutex**, **CreateSemaphore**, or **CreateWaitableTimer** function. If you specify **NULL**, the object gets a default security descriptor. The Access-Control Lists (ACLs) in the default security descriptor for a synchronization object come from the primary or impersonation token of the creator.

To get or set the security descriptor of an event, mutex, semaphore, or waitable timer object, call the **GetNamedSecurityInfo**, **SetNamedSecurityInfo**, **GetSecurityInfo**, or **SetSecurityInfo** functions.

The handles returned by **CreateEvent**, **CreateMutex**, **CreateSemaphore**, and **CreateWaitableTimer** have full access to the new object. When you call the **OpenEvent**, **OpenMutex**, **OpenSemaphore**, and **OpenWaitableTimer** functions, the system checks the requested access rights against the object's security descriptor.

The valid access rights for the interprocess synchronization objects include the standard access rights and some object-specific access rights. The following table lists the standard access rights used by all objects.

Value	Meaning
DELETE(0x00010000L)	Required to delete the object.
READ_CONTROL(0x00020000L)	Required to read information in the security descriptor for the object, not including the information in the SACL. To read or write the SACL, you must request the **ACCESS_SYSTEM_SECURITY**access right. For more information, see SACL Access Right.

Value	Meaning
SYNCHRONIZE(0x00100000L)	The right to use the object for synchronization. This enables a thread to wait until the object is in the signaled state.
WRITE_DAC(0x00040000L)	Required to modify the DACL in the security descriptor for the object.
WRITE_OWNER(0x00080000L)	Required to change the owner in the security descriptor for the object.

The following table lists the object-specific access rights for event objects. These rights are supported in addition to the standard access rights.

Value	Meaning
EVENT_ALL_ACCESS(0x1F0003)	All possible access rights for an event object. Use this right only if your application requires access beyond that granted by the standard access rights and **EVENT_MODIFY_STATE**. Using this access right increases the possibility that your application must be run by an Administrator.
EVENT_MODIFY_STATE(0x0002)	Modify state access, which is required for the **SetEvent**, **ResetEvent** and **PulseEvent** functions.

The following table lists the object-specific access rights for mutex objects. These rights are supported in addition to the standard access rights.

Value	Meaning
MUTEX_ALL_ACCESS(0x1F0001)	All possible access rights for a mutex object. Use this right only if your application requires access beyond that granted by the standard access rights.

Value	Meaning
	Using this access right increases the possibility that your application must be run by an Administrator.
MUTEX_MODIFY_STATE(0x0001)	Reserved for future use.

The following table lists the object-specific access rights for semaphore objects. These rights are supported in addition to the standard access rights.

Value	Meaning
SEMAPHORE_ALL_ACCESS(0x1F0003)	All possible access rights for a semaphore object. Use this right only if your application requires access beyond that granted by the standard access rights and **SEMAPHORE_MODIFY_STATE**. Using this access right increases the possibility that your application must be run by an Administrator.
SEMAPHORE_MODIFY_STATE(0x0002)	Modify state access, which is required for the **ReleaseSemaphore** function.

The following table lists the object-specific access rights for waitable timer objects. These rights are supported in addition to the standard access rights.

Value	Meaning
TIMER_ALL_ACCESS(0x1F0003)	All possible access rights for a waitable timer object. Use this right only if your application requires access beyond that granted by the standard access rights and **TIMER_MODIFY_STATE**. Using this access right increases the possibility that your application must be run by an Administrator.

Value	Meaning
TIMER_MODIFY_STATE(0x0002)	Modify state access, which is required for the **SetWaitableTimer** and **CancelWaitableTimer** functions.
TIMER_QUERY_STATE(0x0001)	Reserved for future use.

To read or write the SACL of an interprocess synchronization object, you must request the **ACCESS_SYSTEM_SECURITY** access right. For more information, see Access-Control Lists (ACLs) and SACL Access Right.

Interprocess Synchronization

Multiple processes can have handles to the same event, mutex, semaphore, or timer object, so these objects can be used to accomplish interprocess synchronization. The process that creates an object can use the handle returned by the creation function (**CreateEvent**, **CreateMutex**, **CreateSemaphore**, or **CreateWaitableTimer**). Other processes can open a handle to the object by using its name, or through inheritance or duplication. For more information, see the following topics:

- Object Names
- Object Inheritance
- Object Duplication

Object Names

Named objects provide an easy way for processes to share object handles. After a process has created a named event, mutex, semaphore, or timer object, other processes can use the name to call the appropriate function (**OpenEvent**, **OpenMutex**, **OpenSemaphore**, or **OpenWaitableTimer**) to open a handle to the object. Name comparison is case sensitive.

The names of event, semaphore, mutex, waitable timer, file-mapping, and job objects share the same namespace. If you try to create an object using a name that is in use by an object of another type, the function fails and **GetLastError**returns **ERROR_INVALID_HANDLE**. Therefore, when creating named objects, use unique names and be sure to check function return values for duplicate-name errors.

If you try to create an object using a name that is in use by an object of same type, the function succeeds, returning a handle to the existing object, and **GetLastError** returns **ERROR_ALREADY_EXISTS**. For example, if the name specified in a call to the **CreateMutex** function matches the name of an existing mutex object, the function returns a handle to the existing object. In this case, the call to **CreateMutex** is equivalent to a call to the **OpenMutex** function. Having multiple processes use **CreateMutex** for the same mutex is therefore equivalent to having one process that calls **CreateMutex** while the other processes call **OpenMutex**, except that it eliminates the need to ensure that the creating process is started first. When using this

technique for mutex objects, however, none of the calling processes should request immediate ownership of the mutex. If multiple processes do request immediate ownership, it can be difficult to predict which process actually gets the initial ownership.

A Terminal Services environment has a global namespace for events, semaphores, mutexes, waitable timers, file-mapping objects, and job objects. In addition, each Terminal Services client session has its own separate namespace for these objects. Terminal Services client processes can use object names with a "Global\" or "Local\" prefix to explicitly create an object in the global or session namespace. For more information, see Kernel Object Namespaces. Fast user switching is implemented using Terminal Services sessions (each user logs into a different session). Kernel object names must follow the guidelines outlined for Terminal Services so that applications can support multiple users.

Synchronization objects can be created in a private namespace. For more information, see Object Namespaces.

Related topics

Using Named Objects

Object Namespaces

An *object namespace* protects named objects from unauthorized access. Creating a private namespace enables applications and services to build a more secure environment.

A process can create a private namespace using the **CreatePrivateNamespace**function. This function requires that you specify a *boundary* that defines how the objects in the namespace are to be isolated. The caller must be within the specified boundary for the create operation to succeed. To specify a boundary, use the **CreateBoundaryDescriptor** and **AddSIDToBoundaryDescriptor** functions.

The *lpAliasPrefix* parameter of **CreatePrivateNamespace** serves as the name of the namespace. Each namespace is uniquely identified by its name and boundaries. The system supports multiple private namespaces with the same name, as long as they specify different boundaries.

Suppose that a process requests the creation of a namespace, NS1, that defines a boundary containing two elements: the administrator SID and the current session number. The namespace is created if the process is running under the Administrator account in the specified session. Another process can access this namespace using the **OpenPrivateNamespace** function. Both the specified name and boundary must match if this process is to open the namespace created by the first process. Note that a process can open an existing namespace even if it is not within the boundary unless the creator restricted access to the namespace using the *lpPrivateNamespaceAttributes* parameter.

The objects that are created in this namespace have names that are of the form *prefix**objectname*. The prefix is the namespace name specified by the *lpAliasPrefix*parameter of **CreatePrivateNamespace**. For example, to create an event object named MyEvent in the NS1 namespace, call the **CreateEvent** function with the *lpName* parameter set to NS1\MyEvent.

The process that created the namespace can use the **ClosePrivateNamespace**function to close the handle to the namespace. The handle is also closed when the process that created the namespace terminates. After the namespace handle is closed, subsequent calls to **OpenPrivateNamespace** fail, but all operations on objects in the namespace succeed.

Related topics

Kernel Object Namespaces

Object Inheritance

When you create a process with the **CreateProcess** function, you can specify that the process inherit handles to mutex, event, semaphore, or timer objects using the **SECURITY_ATTRIBUTES** structure. The handle inherited by the process has the same access to the object as the original handle. The inherited handle appears in the handle table of the created process, but you must communicate the handle value to the created process. You can do this by specifying the value as a command-line argument when you call **CreateProcess**. The created process then uses the **GetCommandLine** function to retrieve the command-line string and convert the handle argument into a usable handle. For more information, see Inheritance.

Object Duplication

The **DuplicateHandle** function creates a duplicate handle that can be used by another specified process. This method of sharing object handles is more complex than using named objects or inheritance. It requires communication between the creating process and the process into which the handle is duplicated. The necessary information (the handle value and process identifier) can be communicated by any of the interprocess communication methods, such as named pipes or named shared memory.

Synchronization and Multiprocessor Issues

Applications may encounter problems when run on multiprocessor systems due to assumptions they make which are valid only on single-processor systems.

Thread Priorities

Consider a program with two threads, one with a higher priority than the other. On a single-processor system, the higher priority thread will not relinquish control to the lower priority thread because the scheduler gives preference to higher priority threads. On a multiprocessor system, both threads can run simultaneously, each on its own processor.

Applications should synchronize access to data structures to avoid race conditions. Code that assumes that higher priority threads run without interference from lower priority threads will fail on multiprocessor systems.

Memory Ordering

When a processor writes to a memory location, the value is cached to improve performance. Similarly, the processor attempts to satisfy read requests from the cache to improve performance. Furthermore, processors begin to fetch values from memory before they are requested by the application. This can happen as part of speculative execution or due to cache line issues.

CPU caches can be partitioned into banks that can be accessed in parallel. This means that memory operations can be completed out of order. To ensure that memory operations are completed in order, most processors provide memory-barrier instructions. A *full memory barrier* ensures that memory read and write operations that appear before the memory barrier instruction are committed to memory before any memory read and write operations that appear after the memory barrier instruction. A *read memory barrier* orders only the memory read operations and a *write memory barrier* orders only the memory write operations. These instructions also ensure that the compiler disables any optimizations that could reorder memory operations across the barriers.

Processors can support instructions for memory barriers with acquire, release, and fence semantics. These semantics describe the order in which results of an operation become available. With acquire semantics, the results of the operation are available before the

results of any operation that appears after it in code. With release semantics, the results of the operation are available after the results of any operation that appears before it in code. Fence semantics combine acquire and release semantics. The results of an operation with fence semantics are available before those of any operation that appears after it in code and after those of any operation that appears before it.

On Intel Itanium-based systems, the instruction is mf (memory fence) with the following modifiers: acq (acquire) and rel (release). On processors that support SSE2, the instructions are mfence (memory fence), lfence (load fence), and sfence (store fence). For more information, see the documentation for the processor.

The following synchronization functions use the appropriate barriers to ensure memory ordering:

- Functions that enter or leave critical sections
- Functions that signal synchronization objects
- Wait functions
- Interlocked functions

Fixing a Race Condition

The following code has a race condition on a multiprocessor systems because the processor that executes `CacheComputedValue` the first time may write `fValueHasBeenComputed` to main memory before writing `iValue` to main memory. Consequently, a second processor executing `FetchComputedValue` at the same time reads `fValueHasBeenComputed` as **TRUE**, but the new value of `iValue` is still in the first processor's cache and has not been written to memory.

syntaxCopy

```
int iValue;
BOOL fValueHasBeenComputed = FALSE;
extern int ComputeValue();

void CacheComputedValue()
{
  if (!fValueHasBeenComputed)
  {
    iValue = ComputeValue();
    fValueHasBeenComputed = TRUE;
  }
}

BOOL FetchComputedValue(int *piResult)
```

```
{
  if (fValueHasBeenComputed)
  {
    *piResult = iValue;
    return TRUE;
  }

  else return FALSE;
}
```

This race condition above can be repaired by using the **volatile** keyword or the **InterlockedExchange** function to ensure that the value of `iValue` is updated for all processors before the value of `fValueHasBeenComputed` is set to **TRUE**.

With Visual Studio 2005, the compiler uses acquire semantics for read operations on **volatile** variables and release semantics for write operations on **volatile**variables (when supported by the CPU). Therefore, you can correct the example as follows:
syntaxCopy

```
volatile int iValue;
volatile BOOL fValueHasBeenComputed = FALSE;
extern int ComputeValue();

void CacheComputedValue()
{
  if (!fValueHasBeenComputed)
  {
    iValue = ComputeValue();
    fValueHasBeenComputed = TRUE;
  }
}

BOOL FetchComputedValue(int *piResult)
{
  if (fValueHasBeenComputed)
  {
    *piResult = iValue;
    return TRUE;
  }

  else return FALSE;
}
```

With Visual Studio 2003, **volatile** to **volatile** references are ordered; the compiler will not re-order **volatile** variable access. However, these operations could be re-ordered by the processor. Therefore, you can correct the example as follows:
syntaxCopy

```
int iValue;
BOOL fValueHasBeenComputed = FALSE;
extern int ComputeValue();

void CacheComputedValue()
{
  if (InterlockedCompareExchange((LONG*)&fValueHasBeenComputed,
          FALSE, FALSE)==FALSE)
  {
    InterlockedExchange ((LONG*)&iValue, (LONG)ComputeValue());
    InterlockedExchange ((LONG*)&fValueHasBeenComputed, TRUE);
  }
}

BOOL FetchComputedValue(int *piResult)
{
  if (InterlockedCompareExchange((LONG*)&fValueHasBeenComputed,
          TRUE, TRUE)==TRUE)
  {
    InterlockedExchange((LONG*)piResult, (LONG)iValue);
    return TRUE;
  }

  else return FALSE;
}
```

Related topics

Critical Section Objects

Interlocked Variable Access

Wait Functions

Synchronization and Overlapped Input and Output

You can perform either synchronous or asynchronous (also called overlapped) I/O operations on files, named pipes, and serial communications devices.
The **WriteFile**, **ReadFile**, **DeviceIoControl**, **WaitCommEvent**, **ConnectNamedPipe**, and **TransactNamedPipe** functions can be performed either synchronously or asynchronously. The **ReadFileEx** and **WriteFileEx** functions can be performed only asynchronously.

When a function is executed synchronously, it does not return until the operation has been completed. This means that the execution of the calling thread can be blocked for an indefinite period while it waits for a time-consuming operation to finish. Functions called for overlapped operation can return immediately, even though the operation has not been completed. This enables a time-consuming I/O operation to be executed in the background while the calling thread is free to perform other tasks. For example, a single thread can perform simultaneous I/O operations on different handles, or even simultaneous read and write operations on the same handle.

To synchronize its execution with the completion of the overlapped operation, the calling thread uses the **GetOverlappedResult** function,
the **GetOverlappedResultEx** function, or one of the wait functions to determine when the overlapped operation has been completed. You can also use
the **HasOverlappedIoCompleted** macro to poll for completion.

To cancel all pending asynchronous I/O operations, use the **CancelIoEx** function and provide an **OVERLAPPED** structure that specifies the request to cancel. Use
the **CancelIo** function to cancel pending asynchronous I/O operations issued by the calling thread for the specified file handle.

Overlapped operations require a file, named pipe, or communications device that was created with the **FILE_FLAG_OVERLAPPED** flag. When a thread calls a function (such as the **ReadFile** function) to perform an overlapped operation, the calling thread must specify a pointer to an **OVERLAPPED** structure. (If this pointer is **NULL**, the function return value may incorrectly indicate that the operation completed.) All of the members of the **OVERLAPPED** structure must be initialized to zero unless an event will be used to signal completion of an I/O operation. If an event is used, the **hEvent** member of the **OVERLAPPED** structure specifies a handle to the allocated event object. The system sets the state of the event object to nonsignaled when a call to the I/O function returns

before the operation has been completed. The system sets the state of the event object to signaled when the operation has been completed. An event is needed only if there will be more than one outstanding I/O operation at the same time. If an event is not used, each completed I/O operation will signal the file, named pipe, or communications device.

When a function is called to perform an overlapped operation, the operation might be completed before the function returns. When this happens, the results are handled as if the operation had been performed synchronously. If the operation was not completed, however, the function's return value is **FALSE**, and the **GetLastError** function returns **ERROR_IO_PENDING**.

A thread can manage overlapped operations by either of two methods:

- Use the **GetOverlappedResult** or **GetOverlappedResultEx** function to wait for the overlapped operation to be completed. If **GetOverlappedResultEx** is used, the calling thread can specify a timeout for the overlapped operation or perform an alertable wait.
- Specify a handle to the **OVERLAPPED** structure's manual-reset event object in one of the wait functions and then, after the wait function returns, call **GetOverlappedResult** or **GetOverlappedResultEx**. The function returns the results of the completed overlapped operation, and for functions in which such information is appropriate, it reports the actual number of bytes that were transferred.

When performing multiple simultaneous overlapped operations on a single thread, the calling thread must specify an **OVERLAPPED** structure for each operation. Each **OVERLAPPED** structure must specify a handle to a different manual-reset event object. To wait for any one of the overlapped operations to be completed, the thread specifies all the manual-reset event handles as wait criteria in one of the multiple-object wait functions. The return value of the multiple-object wait function indicates which manual-reset event object was signaled, so the thread can determine which overlapped operation caused the wait operation to be completed.

It is safer to use a separate event object for each overlapped operation, rather than specify no event object or reuse the same event object for multiple operations. If no event object is specified in the **OVERLAPPED** structure, the system signals the state of the file, named pipe, or communications device when the overlapped operation has been completed. Thus, you can specify these handles as synchronization objects in a wait function, though their use for this purpose can be difficult to manage because, when performing simultaneous overlapped operations on the same file, named pipe, or communications device, there is no way to know which operation caused the object's state to be signaled.

A thread should not reuse an event with the assumption that the event will be signaled only by that thread's overlapped operation. An event is signaled on the same thread as the overlapped operation that is completing. Using the same event on multiple threads can lead to a race condition in which the event is signaled correctly for the thread whose operation completes first and prematurely for other threads using that event. Then, when the next overlapped operation completes, the event is signaled again for all threads using that event, and so on until all overlapped operations are complete.

For examples that illustrate the use of overlapped operations, completion routines, and the **GetOverlappedResult** function, see Using Pipes.

**Windows Vista, Windows Server 2003 and Windows XP: **

Be careful when reusing **OVERLAPPED** structures. If **OVERLAPPED** structures are reused on multiple threads and **GetOverlappedResult** is called with the *bWait*parameter set to **TRUE**, the calling thread must ensure that the associated event is signaled before reusing the structure. This can be accomplished by using the **WaitForSingleObject** function after calling **GetOverlappedResult** to force the thread to wait until the operation completes. Note that the event object must be a manual-reset event object. If an autoreset event object is used, calling **GetOverlappedResult** with the *bWait* parameter set to **TRUE** causes the function to be blocked indefinitely. This behavior changed starting with Windows 7 and Windows Server 2008 R2 for applications that specify Windows 7 as the supported operating system in the application manifest. For more information see Application Manifests.

Related topics

I/O Concepts

Asynchronous Procedure Calls

An *asynchronous procedure call* (APC) is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function. An APC generated by the system is called a *kernel-mode APC*. An APC generated by an application is called a *user-mode APC*. A thread must be in an alertable state to run a user-mode APC.

Each thread has its own APC queue. An application queues an APC to a thread by calling the **QueueUserAPC** function. The calling thread specifies the address of an APC function in the call to **QueueUserAPC**. The queuing of an APC is a request for the thread to call the APC function.

When a user-mode APC is queued, the thread to which it is queued is not directed to call the APC function unless it is in an alertable state. A thread enters an alertable state when it calls
the **SleepEx**, **SignalObjectAndWait**, **MsgWaitForMultipleObjectsEx**, **WaitForMultipleObjectsEx**, or **WaitForSingleObjectEx** function. If the wait is satisfied before the APC is queued, the thread is no longer in an alertable wait state so the APC function will not be executed. However, the APC is still queued, so the APC function will be executed when the thread calls another alertable wait function.

The **ReadFileEx**, **SetWaitableTimer**, **SetWaitableTimerEx**, and **WriteFileEx**functions are implemented using an APC as the completion notification callback mechanism.

If you are using a thread pool, note that APCs do not work as well as other signaling mechanisms because the system controls the lifetime of thread pool threads, so it is possible for a thread to be terminated before the notification is delivered. Instead of using an APC-based signaling mechanism such as the *pfnCompletionRoutine* parameter of **SetWaitableTimer** or **SetWaitableTimerEx**, use a waitable object such as a timer created with **CreateThreadpoolTimer**. For I/O, use an I/O completion object created with **CreateThreadpoolIo** or an *hEvent*-based **OVERLAPPED** structure where the event can be passed to the **SetThreadpoolWait** function.

Synchronization Internals

When an I/O request is issued, a structure is allocated to represent the request. This structure is called an I/O request packet (IRP). With synchronous I/O, the thread builds the IRP, sends it to the device stack, and waits in the kernel for the IRP to complete. With asynchronous I/O, the thread builds the IRP and sends it to the device stack. The

stack might complete the IRP immediately, or it might return a pending status indicating that the request is in progress. When this happens, the IRP is still associated with the thread, so it will be canceled if the thread terminates or calls a function such as **Cancello**. In the meantime, the thread can continue to perform other tasks while the device stack continues to process the IRP.

There are several ways that the system can indicate that the IRP has completed:

- Update the overlapped structure with the result of the operation so the thread can poll to determine whether the operation has completed.
- Signal the event in the overlapped structure so a thread can synchronize on the event and be woken when the operation completes.
- Queue the IRP to the thread's pending APC so that the thread will execute the APC routine when it enters an alertable wait state and return from the wait operation with a status indicating that it executed one or more APC routines.
- Queue the IRP to an I/O completion port, where it will be executed by the next thread that waits on the completion port.

Threads that wait on an I/O completion port do not wait in an alertable state. Therefore, if those threads issue IRPs that are set to complete as APCs to the thread, those IPC completions will not occur in a timely manner; they will occur only if the thread picks up a request from the I/O completion port and then happens to enter an alertable wait.

Related topics

Using a Waitable Timer with an Asynchronous Procedure Call

Critical Section Objects

A *critical section object* provides synchronization similar to that provided by a mutex object, except that a critical section can be used only by the threads of a single process. Critical section objects cannot be shared across processes.

Event, mutex, and semaphore objects can also be used in a single-process application, but critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization (a processor-specific test and set instruction). Like a mutex object, a critical section object can be owned by only one thread at a time, which makes it useful for protecting a shared resource from simultaneous access. Unlike a mutex object, there is no way to tell whether a critical section has been abandoned.

Starting with Windows Server 2003 with Service Pack 1 (SP1), threads waiting on a critical section do not acquire the critical section on a first-come, first-serve basis. This change increases performance significantly for most code. However, some applications depend on first-in, first-out (FIFO) ordering and may perform poorly or not at all on current versions of Windows (for example, applications that have been using critical sections as a rate-limiter). To ensure that your code continues to work correctly, you may need to add an additional level of synchronization. For example, suppose you have a producer thread and a consumer thread that are using a critical section object to synchronize their work. Create two event objects, one for each thread to use to signal that it is ready for the other thread to proceed. The consumer thread will wait for the producer to signal its event before entering the critical section, and the producer thread will wait for the consumer thread to signal its event before entering the critical section. After each thread leaves the critical section, it signals its event to release the other thread.

Windows Server 2003 and Windows XP: Threads that are waiting on a critical section are added to a wait queue; they are woken and generally acquire the critical section in the order in which they were added to the queue. However, if threads are added to this queue at a fast enough rate, performance can be degraded because of the time it takes to awaken each waiting thread.

The process is responsible for allocating the memory used by a critical section. Typically, this is done by simply declaring a variable of type **CRITICAL_SECTION**. Before the threads of the process can use it, initialize the critical section by using the **InitializeCriticalSection** or **InitializeCriticalSectionAndSpinCount** function.

A thread uses the **EnterCriticalSection** or **TryEnterCriticalSection** function to request ownership of a critical section. It uses the **LeaveCriticalSection** function to release ownership of a critical section. If the critical section object is currently owned by another thread, **EnterCriticalSection** waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the wait functionsaccept a specified time-out interval. The **TryEnterCriticalSection** function attempts to enter a critical section without blocking the calling thread.

When a thread owns a critical section, it can make additional calls to **EnterCriticalSection** or **TryEnterCriticalSection** without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns. To release its ownership, the thread must call **LeaveCriticalSection**one time for each time that it entered the critical section. There is no guarantee about the order in which waiting threads will acquire ownership of the critical section.

A thread uses the **InitializeCriticalSectionAndSpinCount** or **SetCriticalSectionSpinCount** function to specify a spin count for the critical section object. Spinning means that when a thread tries to acquire a critical section that is locked, the thread enters a loop, checks to see if the lock is released, and if the lock is not released, the thread goes to sleep. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0 (zero). On multiprocessor systems, if the critical section is unavailable, the calling thread spins *dwSpinCount* times before performing a wait operation on a semaphore that is associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that are allocated when the critical section object is initialized. After this function is called, the critical section object cannot be used for synchronization.

When a critical section object is owned, the only other threads affected are the threads that are waiting for ownership in a call to **EnterCriticalSection**. Threads that are not waiting are free to continue running.

Related topics

Mutex Objects

Using Critical Section Objects

Condition Variables

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs. Condition variables are user-mode objects that cannot be shared across processes.

Condition variables enable threads to atomically release a lock and enter the sleeping state. They can be used with critical sections or slim reader/writer (SRW) locks. Condition variables support operations that "wake one" or "wake all" waiting threads. After a thread is woken, it re-acquires the lock it released when the thread entered the sleeping state.

Note that the caller must allocate a **CONDITION_VARIABLE** structure and initialize it by either calling **InitializeConditionVariable** (to initialize the structure dynamically) or assign the constant **CONDITION_VARIABLE_INIT** to the structure variable (to initialize the structure statically).

Windows Server 2003 and Windows XP: Condition variables are not supported.

The following are the condition variable functions.

Condition variable function	Description
InitializeConditionVariable	Initializes a condition variable.
SleepConditionVariableCS	Sleeps on the specified condition variable and releases the specified critical section as an atomic operation.
SleepConditionVariableSRW	Sleeps on the specified condition variable and releases the specified SRW lock as an atomic operation.
WakeAllConditionVariable	Wakes all threads waiting on the specified condition variable.
WakeConditionVariable	Wakes a single thread waiting on the specified condition variable.

The following pseudocode demonstrates the typical usage pattern of condition variables.
syntaxCopy

```
CRITICAL_SECTION CritSection;
CONDITION_VARIABLE ConditionVar;

void PerformOperationOnSharedData()
{
   EnterCriticalSection(&CritSection);

   // Wait until the predicate is TRUE

   while( TestPredicate() == FALSE )
   {
      SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE);
   }

   // The data can be changed safely because we own the critical
   // section and the predicate is TRUE

   ChangeSharedData();

   LeaveCriticalSection(&CritSection);

   // If necessary, signal the condition variable by calling
   // WakeConditionVariable or WakeAllConditionVariable so other
   // threads can wake
}
```

For example, in an implementation of a reader/writer lock, the `TestPredicate`function would verify that the current lock request is compatible with the existing owners. If it is, acquire the lock; otherwise, sleep. For a more detailed example, see Using Condition Variables.

Condition variables are subject to spurious wakeups (those not associated with an explicit wake) and stolen wakeups (another thread manages to run before the woken thread). Therefore, you should recheck a predicate (typically in a **while**loop) after a sleep operation returns.

You can wake other threads
using **WakeConditionVariable** or **WakeAllConditionVariable** either inside or outside

the lock associated with the condition variable. It is usually better to release the lock before waking other threads to reduce the number of context switches.

It is often convenient to use more than one condition variable with the same lock. For example, an implementation of a reader/writer lock might use a single critical section but separate condition variables for readers and writers.

Related topics

Using Condition Variables

Slim Reader/Writer (SRW) Locks

Slim reader/writer (SRW) locks enable the threads of a single process to access shared resources; they are optimized for speed and occupy very little memory. Slim reader-writer locks cannot be shared across processes.

Reader threads read data from a shared resource whereas writer threads write data to a shared resource. When multiple threads are reading and writing using a shared resource, exclusive locks such as a critical section or mutex can become a bottleneck if the reader threads run continuously but write operations are rare.

SRW locks provide two modes in which threads can access a shared resource:

- **Shared mode**, which grants shared read-only access to multiple reader threads, which enables them to read data from the shared resource concurrently. If read operations exceed write operations, this concurrency increases performance and throughput compared to critical sections.
- **Exclusive mode**, which grants read/write access to one writer thread at a time. When the lock has been acquired in exclusive mode, no other thread can access the shared resource until the writer releases the lock.

A single SRW lock can be acquired in either mode; reader threads can acquire it in shared mode whereas writer threads can acquire it in exclusive mode. There is no guarantee about the order in which threads that request ownership will be granted ownership; SRW locks are neither fair nor FIFO.

An SRW lock is the size of a pointer. The advantage is that it is fast to update the lock state. The disadvantage is that very little state information can be stored, so SRW locks cannot be acquired recursively. In addition, a thread that owns an SRW lock in shared mode cannot upgrade its ownership of the lock to exclusive mode.

The caller must allocate an SRWLOCK structure and initialize it by either calling **InitializeSRWLock** (to initialize the structure dynamically) or assign the constant **SRWLOCK_INIT** to the structure variable (to initialize the structure statically).

The following are the SRW lock functions.

SRW lock function	Description
AcquireSRWLockExclusive	Acquires an SRW lock in exclusive mode.

SRW lock function	Description
AcquireSRWLockShared	Acquires an SRW lock in shared mode.
InitializeSRWLock	Initialize an SRW lock.
ReleaseSRWLockExclusive	Releases an SRW lock that was opened in exclusive mode.
ReleaseSRWLockShared	Releases an SRW lock that was opened in shared mode.
SleepConditionVariableSRW	Sleeps on the specified condition variable and releases the specified lock as an atomic operation.
TryAcquireSRWLockExclusive	Attempts to acquire a slim reader/writer (SRW) lock in exclusive mode. If the call is successful, the calling thread takes ownership of the lock.
TryAcquireSRWLockShared	Attempts to acquire a slim reader/writer (SRW) lock in shared mode. If the call is successful, the calling thread takes ownership of the lock.

One-Time Initialization

Components are often designed to perform initialization tasks when they are first called, rather than when they are loaded. The one-time initialization functions ensure that this initialization occurs only once, even when multiple threads may attempt the initialization.

Windows Server 2003 and Windows XP: Applications must provide their own synchronization for one-time initialization by using the interlocked functions or other synchronization mechanism. The one-time initialization functions are available starting with Windows Vista and Windows Server 2008.

The one-time initialization functions provide significant advantages to ensure that only one thread performs the initialization:

- They are optimized for speed.
- They create the appropriate barriers on processor architectures that require them.
- They support both locked and parallel initialization.
- They avoid internal locking so the code can operate asynchronously or synchronously.

The system manages the initialization process through an opaque **INIT_ONCE**structure that contains data and state information. The caller allocates this structure and initializes it by either calling **InitOnceInitialize** (to initialize the structure dynamically) or assigning the constant **INIT_ONCE_STATIC_INIT** to the structure variable (to initialize the structure statically). Initially, the data stored in the one-time initialization structure is NULL and its state is uninitialized.

One-time initialization structures cannot be shared across processes.

The thread that performs the initialization can optionally set a context that is available to the caller after initialization is complete. The context can be a synchronization object or it can be a value or data structure. If the context is a value, its low-order **INIT_ONCE_CTX_RESERVED_BITS** must be zero. If the context is a data structure, the data structure must be **DWORD**-aligned. The context is returned to the caller in the *lpContext* output parameter of the **InitOnceBeginInitialize** or **InitOnceExecuteOnce** function.

One-time initialization can be performed synchronously or asynchronously. An optional callback function can be used for synchronous one-time initialization.

Synchronous One-time Initialization

The following steps describe synchronous one-time initialization that does not use a callback function.

1. The first thread to call the **InitOnceBeginInitialize** function successfully causes one-time initialization to begin. For synchronous one-time initialization, **InitOnceBeginInitialize** must be called without the **INIT_ONCE_ASYNC** flag.
2. Subsequent threads that attempt initialization are blocked until the first thread either completes initialization or fails. If the first thread fails, the next thread is allowed to attempt the initialization, and so on.
3. When initialization is finished, the thread calls the **InitOnceComplete**function. The thread can optionally create a synchronization object (or other context data) and specify it in the *lpContext* parameter of the **InitOnceComplete** function.
4. If the initialization succeeds, the state of the one-time initialization structure is changed to initialized and the *lpContext* handle (if any) is stored in the initialization structure. Subsequent initialization attempts return this context data. If the initialization fails, the data is **NULL**.

The following steps describe synchronous one-time initialization that uses a callback function.

1. The first thread to successfully call the **InitOnceExecuteOnce** function passes a pointer to an application-defined *InitOnceCallback* callback function and any data required by the callback function. If the call succeeds, the *InitOnceCallback* callback function executes.
2. Subsequent threads that attempt initialization are blocked until the first thread either completes initialization or fails. If the first thread fails, the next thread is allowed to attempt the initialization, and so on.
3. When initialization is finished, the callback function returns. The callback function can optionally create a synchronization object (or other context data) and specify it in its *Context* output parameter.
4. If the initialization succeeds, the state of the one-time initialization structure is changed to initialized and the *Context* handle (if any) is stored in the initialization structure. Subsequent initialization attempts return this context data. If the initialization fails, the data is **NULL**.

Asynchronous One-time Initialization

The following steps describe asynchronous one-time initialization.

1. If multiple threads simultaneously attempt to begin initialization by calling **InitOnceBeginInitialize** with **INIT_ONCE_ASYNC**, the function succeeds for all of

the threads with the *fPending* parameter set to **TRUE**. Only one thread will actually succeed at initialization; other concurrent attempts do not change the initialization state.

2. When **InitOnceBeginInitialize** returns, the *fPending* parameter indicates the initialization status:
    - If *fPending* is **FALSE**, one thread has succeeded at initialization. Other threads should clean up any context data they have created and use the context data in the *lpContext* output parameter of **InitOnceBeginInitialize**.
    - If *fPending* is **TRUE**, initialization has not yet completed and other threads should continue.
3. Each thread calls the **InitOnceComplete** function. The thread can optionally create a synchronization object (or other context data) and specify it in the *lpContext* parameter of **InitOnceComplete**.
4. When **InitOnceComplete** returns, its return value indicates whether the calling thread succeeded at initialization.
    - If **InitOnceComplete** succeeds, the calling thread has succeeded at initialization. The state of the one-time initialization structure is changed to initialized and the *lpContext* handle (if any) is stored in the initialization structure.
    - If **InitOnceComplete** fails, another thread has succeeded at initialization. The calling thread should clean up any context data it has created and call **InitOnceBeginInitialize** with **INIT_ONCE_CHECK_ONLY**to retrieve any context data stored in the one-time initialization structure.

Related topics

Using One-Time Initialization

Interlocked Variable Access

Applications must synchronize access to variables that are shared by multiple threads. Applications must also ensure that operations on these variables are performed atomically (performed in their entirety or not at all.)

Simple reads and writes to properly-aligned 32-bit variables are atomic operations. In other words, you will not end up with only one portion of the variable updated; all bits are updated in an atomic fashion. However, access is not guaranteed to be synchronized. If two threads are reading and writing from the same variable, you cannot determine if one thread will perform its read operation before the other performs its write operation.

Simple reads and writes to properly aligned 64-bit variables are atomic on 64-bit Windows. Reads and writes to 64-bit values are not guaranteed to be atomic on 32-bit Windows. Reads and writes to variables of other sizes are not guaranteed to be atomic on any platform.

The Interlocked API

The interlocked functions provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. They also perform operations on variables in an atomic manner. The threads of different processes can use these functions if the variable is in shared memory.

The **InterlockedIncrement** and **InterlockedDecrement** functions combine the steps involved in incrementing or decrementing a variable into an atomic operation. This feature is useful in a multitasking operating system, in which the system can interrupt one thread's execution to grant a slice of processor time to another thread. Without such synchronization, two threads could read the same value, increment it by 1, and store the new value for a total increase of 1 instead of 2. The interlocked variable-access functions protect against this kind of error.

The **InterlockedExchange** and **InterlockedExchangePointer** functions atomically exchange the values of the specified variables. The **InterlockedExchangeAdd**function combines two operations: adding two variables together and storing the result in one of the variables.

The **InterlockedCompareExchange**, **InterlockedCompare64Exchange128**, and **InterlockedCompareExchangePointer** functions combine two operations:

comparing two values and storing a third value in one of the variables, based on the outcome of the comparison.

The **InterlockedAnd**, **InterlockedOr**, and **InterlockedXor** functions atomically perform AND, OR, and XOR operations, respectively.

There are functions that are specifically designed to perform interlocked variable access on 64-bit memory values and addresses, and are optimized for use on 64-bit Windows. Each of these functions contains "64" in the name; for example, **InterlockedDecrement64** and **InterlockedCompareExchangeAcquire64**.

Most of the interlocked functions provide full memory barriers on all Windows platforms. There are also functions that combine the basic interlocked variable access operations with the acquire and release memory ordering semantics supported by certain processors. Each of these functions contains the word "Acquire" or "Release" in their names; for example, **InterlockedDecrementAcquire**and **InterlockedDecrementRelease**. Acquire memory semantics specify that the memory operation being performed by the current thread will be visible before any other memory operations are attempted. Release memory semantics specify that the memory operation being performed by the current thread will be visible after all other memory operations have been completed. These semantics allow you to force memory operations to be performed in a specific order. Use acquire semantics when entering a protected region and release semantics when leaving it.

Related topics

Compiler Intrinsics

Synchronization and Multiprocessor Issues

Interlocked Singly Linked Lists

An *interlocked singly linked list* (SList) eases the task of insertion and deletion from a linked list. SLists are implemented using a nonblocking algorithm to provide atomic synchronization, increase system performance, and avoid problems such as priority inversion and lock convoys.

SLists are straightforward to implement and use in 32-bit code. However, it is challenging to implement them in 64-bit code because the amount of data exchangeable by the native interlocked exchange primitives is not double the address size, as it is in 32-bit code. Therefore, SLists enable porting high-end scalable algorithms to Windows.

Windows 8: Starting in Windows 8 the appropriate native interlocked exchange primitives are available for 64-bit code, for example **InterlockedCompare64Exchange128**.

Applications can use SLists by calling the **InitializeSListHead** function to initialize the head of the list. To insert items into the list, use the **InterlockedPushEntrySList**function. To delete items from the list, use the **InterlockedPopEntrySList** function.

All list items must be aligned on a **MEMORY_ALLOCATION_ALIGNMENT**boundary. Unaligned items can cause unpredictable results. See **_aligned_malloc**.

For an example, see Using Singly Linked Lists.

The following table lists the SList functions.

Function	Description
InitializeSListHead	Initializes the head of a singly linked list.
InterlockedFlushSList	Flushes the entire list of items in a singly linked list.
InterlockedPopEntrySList	Removes an item from the front of a singly linked list.
InterlockedPushEntrySList	Inserts an item at the front of a singly linked list.

Function	Description
InterlockedPushListSList	Inserts a singly-linked list at the front of another singly linked list.
InterlockedPushListSListEx	Inserts a singly-linked list at the front of another singly linked list. This version of the method does not use the **__fastcall** calling convention.
RtlFirstEntrySList	Retrieves the first entry in a singly linked list.
QueryDepthSList	Retrieves the number of entries in the specified singly linked list.

Timer Queues

The **CreateTimerQueue** function creates a queue for timers. Timers in this queue, known as *timer-queue timers*, are lightweight objects that enable you to specify a callback function to be called when the specified due time arrives. The wait operation is performed by a thread in the thread pool.

To add a timer to the queue, call the **CreateTimerQueueTimer** function. To update a timer-queue timer, call the **ChangeTimerQueueTimer** function. You can specify a callback function to be executed by a worker thread from the thread pool when the timer expires.

To cancel a pending timer, call the **DeleteTimerQueueTimer** function. When you are finished with the queue of timers, call the **DeleteTimerQueueEx** function to delete the timer queue. Any pending timers in the queue are canceled and deleted.

Related topics

Using Timer Queues

Synchronization Barriers

A synchronization barrier enables multiple threads to wait until all threads have all reached a particular point of execution before any thread continues. Synchronization barriers cannot be shared across processes.

Synchronization barriers are useful for phased computations, in which threads executing the same code in parallel must all complete one phase before moving on to the next.

To create a synchronization barrier, call the **InitializeSynchronizationBarrier**function and specify a maximum number of threads and how many times a thread should spin before it blocks. Then launch the threads that will use the barrier. After each thread finishes its work, it calls **EnterSynchronizationBarrier** to wait at the barrier.
The **EnterSynchronizationBarrier** function blocks each thread until the number of threads blocked in the barrier reaches the maximum thread count for the barrier, at which point **EnterSynchronizationBarrier** unblocks all the threads.
The **EnterSynchronizationBarrier** function returns **TRUE** for exactly one of the threads that entered the barrier, and returns **FALSE** for all other threads.

To release a synchronization barrier when it is no longer needed,
call **DeleteSynchronizationBarrier**. It is safe to call this function immediately after calling **EnterSynchronizationBarrier** because that function ensures that all threads have finished using the barrier before it is released.

If a synchronization barrier will never be deleted, threads can specify
the **SYNCHRONIZATION_BARRIER_FLAGS_NO_DELETE** flag when they enter the barrier. All threads using the barrier must specify this flag; if any thread does not, the flag is ignored. This flag causes the function to skip the extra work required for deletion safety, which can improve performance. Note that deleting a barrier while this flag is in effect may result in an invalid handle access and one or more permanently blocked threads.

Using Synchronization

The following examples demonstrate how to use the synchronization objects:

- Waiting for multiple objects
- Using named objects
- Using event objects
- Using mutex objects
- Using semaphore objects
- Using waitable timer objects
- Using waitable timers with an asynchronous procedure call
- Using critical section objects
- Using condition variables
- Using one-time initialization
- Using singly linked lists
- Using timer queues

Waiting for Multiple Objects

The following example uses the **CreateEvent** function to create two event objects and the **CreateThread** function to create a thread. It then uses the **WaitForMultipleObjects** function to wait for the thread to set the state of one of the objects to signaled using the **SetEvent** function.

For an example that waits for a single object, see Using Mutex Objects.

C++Copy

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

HANDLE ghEvents[2];

DWORD WINAPI ThreadProc( LPVOID );

int main( void )
{
    HANDLE hThread;
    DWORD i, dwEvent, dwThreadID;

    // Create two event objects

    for (i = 0; i < 2; i++)
    {
        ghEvents[i] = CreateEvent(
            NULL,   // default security attributes
            FALSE,  // auto-reset event object
            FALSE,  // initial state is nonsignaled
            NULL);  // unnamed object

        if (ghEvents[i] == NULL)
        {
            printf("CreateEvent error: %d\n", GetLastError() );
            ExitProcess(0);
        }
    }

    // Create a thread

    hThread = CreateThread(
                 NULL,         // default security attributes
                 0,            // default stack size
```

```
            (LPTHREAD_START_ROUTINE) ThreadProc,
            NULL,         // no thread function arguments
            0,            // default creation flags
            &dwThreadID); // receive thread identifier

if( hThread == NULL )
{
    printf("CreateThread error: %d\n", GetLastError());
    return 1;
}

// Wait for the thread to signal one of the event objects

dwEvent = WaitForMultipleObjects(
    2,           // number of objects in array
    ghEvents,     // array of objects
    FALSE,       // wait for any object
    5000);       // five-second wait

// The return value indicates which event is signaled

switch (dwEvent)
{
    // ghEvents[0] was signaled
    case WAIT_OBJECT_0 + 0:
        // TODO: Perform tasks required by this event
        printf("First event was signaled.\n");
        break;

    // ghEvents[1] was signaled
    case WAIT_OBJECT_0 + 1:
        // TODO: Perform tasks required by this event
        printf("Second event was signaled.\n");
        break;

    case WAIT_TIMEOUT:
        printf("Wait timed out.\n");
        break;

    // Return value is invalid.
    default:
        printf("Wait error: %d\n", GetLastError());
        ExitProcess(0);
}
```

```
    // Close event handles

    for (i = 0; i < 2; i++)
        CloseHandle(ghEvents[i]);

    return 0;
}

DWORD WINAPI ThreadProc( LPVOID lpParam )
{

    // lpParam not used in this example
    UNREFERENCED_PARAMETER( lpParam);

    // Set one event to the signaled state

    if ( !SetEvent(ghEvents[0]) )
    {
        printf("SetEvent failed (%d)\n", GetLastError());
        return 1;
    }
    return 0;
}
```

Using Named Objects

The following example illustrates the use of object names by creating and opening a named mutex.

First Process

The first process uses the **CreateMutex** function to create the mutex object. Note that this function succeeds even if there is an existing object with the same name.

C++Copy

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

// This process creates the mutex object.

int main(void)
{
    HANDLE hMutex;

    hMutex = CreateMutex(
        NULL,                        // default security descriptor
        FALSE,                       // mutex not owned
        TEXT("NameOfMutexObject"));  // object name

    if (hMutex == NULL)
        printf("CreateMutex error: %d\n", GetLastError() );
    else
        if ( GetLastError() == ERROR_ALREADY_EXISTS )
            printf("CreateMutex opened an existing mutex\n");
        else printf("CreateMutex created a new mutex.\n");

    // Keep this process around until the second process is run
    _getch();

    CloseHandle(hMutex);

    return 0;
}
```

Second Process

The second process uses the **OpenMutex** function to open a handle to the existing mutex. This function fails if a mutex object with the specified name does not exist. The

access parameter requests full access to the mutex object, which is necessary for the handle to be used in any of the wait functions.

C++Copy

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

// This process opens a handle to a mutex created by another process.

int main(void)
{
    HANDLE hMutex;

    hMutex = OpenMutex(
        MUTEX_ALL_ACCESS,            // request full access
        FALSE,                       // handle not inheritable
        TEXT("NameOfMutexObject"));  // object name

    if (hMutex == NULL)
        printf("OpenMutex error: %d\n", GetLastError() );
    else printf("OpenMutex successfully opened the mutex.\n");

    CloseHandle(hMutex);

    return 0;
}
```

Related topics

[Object Names]

[Using Mutex Objects]

Using Event Objects

Applications can use event objects in a number of situations to notify a waiting thread of the occurrence of an event. For example, overlapped I/O operations on files, named pipes, and communications devices use an event object to signal their completion. For more information about the use of event objects in overlapped I/O operations, see Synchronization and Overlapped Input and Output.

The following example uses event objects to prevent several threads from reading from a shared memory buffer while a master thread is writing to that buffer. First, the master thread uses the **CreateEvent** function to create a manual-reset event object whose initial state is nonsignaled. Then it creates several reader threads. The master thread performs a write operation and then sets the event object to the signaled state when it has finished writing.

Before starting a read operation, each reader thread uses **WaitForSingleObject** to wait for the manual-reset event object to be signaled. When **WaitForSingleObject**returns, this indicates that the main thread is ready for it to begin its read operation.

C++Copy

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

#define THREADCOUNT 4

HANDLE ghWriteEvent;
HANDLE ghThreads[THREADCOUNT];

DWORD WINAPI ThreadProc(LPVOID);

void CreateEventsAndThreads(void)
{
    int i;
    DWORD dwThreadID;

    // Create a manual-reset event object. The write thread sets this
    // object to the signaled state when it finishes writing to a
    // shared buffer.

    ghWriteEvent = CreateEvent(
        NULL,               // default security attributes
        TRUE,               // manual-reset event
        FALSE,              // initial state is nonsignaled
```

```
        TEXT("WriteEvent")  // object name
        );

    if (ghWriteEvent == NULL)
    {
        printf("CreateEvent failed (%d)\n", GetLastError());
        return;
    }

    // Create multiple threads to read from the buffer.

    for(i = 0; i < THREADCOUNT; i++)
    {
        // TODO: More complex scenarios may require use of a parameter
        //   to the thread procedure, such as an event per thread to
        //   be used for synchronization.
        ghThreads[i] = CreateThread(
            NULL,              // default security
            0,                 // default stack size
            ThreadProc,        // name of the thread function
            NULL,              // no thread parameters
            0,                 // default startup flags
            &dwThreadID);

        if (ghThreads[i] == NULL)
        {
            printf("CreateThread failed (%d)\n", GetLastError());
            return;
        }
    }
}

void WriteToBuffer(VOID)
{
    // TODO: Write to the shared buffer.

    printf("Main thread writing to the shared buffer...\n");

    // Set ghWriteEvent to signaled

    if (! SetEvent(ghWriteEvent) )
    {
        printf("SetEvent failed (%d)\n", GetLastError());
        return;
    }
```

```
}

void CloseEvents()
{
    // Close all event handles (currently, only one global handle).

    CloseHandle(ghWriteEvent);
}

int main( void )
{
    DWORD dwWaitResult;

    // TODO: Create the shared buffer

    // Create events and THREADCOUNT threads to read from the buffer

    CreateEventsAndThreads();

    // At this point, the reader threads have started and are most
    // likely waiting for the global event to be signaled. However,
    // it is safe to write to the buffer because the event is a
    // manual-reset event.

    WriteToBuffer();

    printf("Main thread waiting for threads to exit...\n");

    // The handle for each thread is signaled when the thread is
    // terminated.
    dwWaitResult = WaitForMultipleObjects(
        THREADCOUNT,   // number of handles in array
        ghThreads,     // array of thread handles
        TRUE,          // wait until all are signaled
        INFINITE);

    switch (dwWaitResult)
    {
        // All thread objects were signaled
        case WAIT_OBJECT_0:
            printf("All threads ended, cleaning up for application
exit...\n");
            break;

        // An error occurred
```

```
        default:
            printf("WaitForMultipleObjects failed (%d)\n", GetLastError());
            return 1;
    }

    // Close the events to clean up

    CloseEvents();

    return 0;
}

DWORD WINAPI ThreadProc(LPVOID lpParam)
{
    // lpParam not used in this example.
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwWaitResult;

    printf("Thread %d waiting for write event...\n", GetCurrentThreadId());

    dwWaitResult = WaitForSingleObject(
        ghWriteEvent, // event handle
        INFINITE);    // indefinite wait

    switch (dwWaitResult)
    {
        // Event object was signaled
        case WAIT_OBJECT_0:
            //
            // TODO: Read from the shared buffer
            //
            printf("Thread %d reading from buffer\n",
                   GetCurrentThreadId());
            break;

        // An error occurred
        default:
            printf("Wait error (%d)\n", GetLastError());
            return 0;
    }

    // Now that we are done reading the buffer, we could use another
    // event to signal that this thread is no longer reading. This
    // example simply uses the thread handle for synchronization (the
```

```
    // handle is signaled when the thread terminates.)

    printf("Thread %d exiting\n", GetCurrentThreadId());
    return 1;
}
```

Using Mutex Objects

You can use a mutex object to protect a shared resource from simultaneous access by multiple threads or processes. Each thread must wait for ownership of the mutex before it can execute the code that accesses the shared resource. For example, if several threads share access to a database, the threads can use a mutex object to permit only one thread at a time to write to the database.

The following example uses the **CreateMutex** function to create a mutex object and the **CreateThread** function to create worker threads.

When a thread of this process writes to the database, it first requests ownership of the mutex using the **WaitForSingleObject** function. If the thread obtains ownership of the mutex, it writes to the database and then releases its ownership of the mutex using the **ReleaseMutex** function.

This example uses structured exception handling to ensure that the thread properly releases the mutex object. The **__finally** block of code is executed no matter how the **__try** block terminates (unless the **__try** block includes a call to the **TerminateThread** function). This prevents the mutex object from being abandoned inadvertently.

If a mutex is abandoned, the thread that owned the mutex did not properly release it before terminating. In this case, the status of the shared resource is indeterminate, and continuing to use the mutex can obscure a potentially serious error. Some applications might attempt to restore the resource to a consistent state; this example simply returns an error and stops using the mutex. For more information, see Mutex Objects.

C++Copy

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

#define THREADCOUNT 2

HANDLE ghMutex;

DWORD WINAPI WriteToDatabase( LPVOID );

int main( void )
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;
```

```
int i;

// Create a mutex with no initial owner

ghMutex = CreateMutex(
    NULL,              // default security attributes
    FALSE,             // initially not owned
    NULL);             // unnamed mutex

if (ghMutex == NULL)
{
    printf("CreateMutex error: %d\n", GetLastError());
    return 1;
}

// Create worker threads
for( i=0; i < THREADCOUNT; i++ )
{
    aThread[i] = CreateThread(
                 NULL,       // default security attributes
                 0,          // default stack size
                 (LPTHREAD_START_ROUTINE) WriteToDatabase,
                 NULL,       // no thread function arguments
                 0,          // default creation flags
                 &ThreadID); // receive thread identifier

    if( aThread[i] == NULL )
    {
        printf("CreateThread error: %d\n", GetLastError());
        return 1;
    }
}

// Wait for all threads to terminate

WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);

// Close thread and mutex handles

for( i=0; i < THREADCOUNT; i++ )
    CloseHandle(aThread[i]);

CloseHandle(ghMutex);

return 0;
```

```
}

DWORD WINAPI WriteToDatabase( LPVOID lpParam )
{
    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwCount=0, dwWaitResult;

    // Request ownership of mutex.

    while( dwCount < 20 )
    {
        dwWaitResult = WaitForSingleObject(
            ghMutex,    // handle to mutex
            INFINITE);  // no time-out interval

        switch (dwWaitResult)
        {
            // The thread got ownership of the mutex
            case WAIT_OBJECT_0:
                __try {
                    // TODO: Write to the database
                    printf("Thread %d writing to database...\n",
                            GetCurrentThreadId());
                    dwCount++;
                }

                __finally {
                    // Release ownership of the mutex object
                    if (! ReleaseMutex(ghMutex))
                    {
                        // Handle error.
                    }
                }
                break;

            // The thread got ownership of an abandoned mutex
            // The database is in an indeterminate state
            case WAIT_ABANDONED:
                return FALSE;
        }
    }
    return TRUE;
}
```

Using Semaphore Objects

The following example uses a semaphore object to limit the number of threads that can perform a particular task. First, it uses the **CreateSemaphore** function to create the semaphore and to specify initial and maximum counts, then it uses the **CreateThread** function to create the threads.

Before a thread attempts to perform the task, it uses the **WaitForSingleObject**function to determine whether the semaphore's current count permits it to do so. The wait function's time-out parameter is set to zero, so the function returns immediately if the semaphore is in the nonsignaled state. **WaitForSingleObject**decrements the semaphore's count by one.

When a thread completes the task, it uses the **ReleaseSemaphore** function to increment the semaphore's count, thus enabling another waiting thread to perform the task.

C++Copy

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

#define MAX_SEM_COUNT 10
#define THREADCOUNT 12

HANDLE ghSemaphore;

DWORD WINAPI ThreadProc( LPVOID );

int main( void )
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;
    int i;

    // Create a semaphore with initial and max counts of MAX_SEM_COUNT

    ghSemaphore = CreateSemaphore(
        NULL,           // default security attributes
        MAX_SEM_COUNT,  // initial count
        MAX_SEM_COUNT,  // maximum count
        NULL);          // unnamed semaphore

    if (ghSemaphore == NULL)
```

```
    {
        printf("CreateSemaphore error: %d\n", GetLastError());
        return 1;
    }

    // Create worker threads

    for( i=0; i < THREADCOUNT; i++ )
    {
        aThread[i] = CreateThread(
                     NULL,       // default security attributes
                     0,          // default stack size
                     (LPTHREAD_START_ROUTINE) ThreadProc,
                     NULL,       // no thread function arguments
                     0,          // default creation flags
                     &ThreadID); // receive thread identifier

        if( aThread[i] == NULL )
        {
            printf("CreateThread error: %d\n", GetLastError());
            return 1;
        }
    }

    // Wait for all threads to terminate

    WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);

    // Close thread and semaphore handles

    for( i=0; i < THREADCOUNT; i++ )
        CloseHandle(aThread[i]);

    CloseHandle(ghSemaphore);

    return 0;
}

DWORD WINAPI ThreadProc( LPVOID lpParam )
{

    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwWaitResult;
```

```
    BOOL bContinue=TRUE;

    while(bContinue)
    {
        // Try to enter the semaphore gate.

        dwWaitResult = WaitForSingleObject(
            ghSemaphore,   // handle to semaphore
            0L);           // zero-second time-out interval

        switch (dwWaitResult)
        {
            // The semaphore object was signaled.
            case WAIT_OBJECT_0:
                // TODO: Perform task
                printf("Thread %d: wait succeeded\n", GetCurrentThreadId());
                bContinue=FALSE;

                // Simulate thread spending time on task
                Sleep(5);

                // Release the semaphore when task is finished

                if (!ReleaseSemaphore(
                        ghSemaphore,  // handle to semaphore
                        1,            // increase count by one
                        NULL) )       // not interested in previous count
                {
                    printf("ReleaseSemaphore error: %d\n", GetLastError());
                }
                break;

            // The semaphore was nonsignaled, so a time-out occurred.
            case WAIT_TIMEOUT:
                printf("Thread %d: wait timed out\n", GetCurrentThreadId());
                break;
        }
    }
    return TRUE;
}
```

Using Waitable Timer Objects

The following example creates a timer that will be signaled after a 10 second delay. First, the code uses the **CreateWaitableTimer** function to create a waitable timer object. Then it uses the **SetWaitableTimer** function to set the timer. The code uses the **WaitForSingleObject** function to determine when the timer has been signaled.
C++Copy

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

int main()
{
    HANDLE hTimer = NULL;
    LARGE_INTEGER liDueTime;

    liDueTime.QuadPart = -100000000LL;

    // Create an unnamed waitable timer.
    hTimer = CreateWaitableTimer(NULL, TRUE, NULL);
    if (NULL == hTimer)
    {
        printf("CreateWaitableTimer failed (%d)\n", GetLastError());
        return 1;
    }

    printf("Waiting for 10 seconds...\n");

    // Set a timer to wait for 10 seconds.
    if (!SetWaitableTimer(hTimer, &liDueTime, 0, NULL, NULL, 0))
    {
        printf("SetWaitableTimer failed (%d)\n", GetLastError());
        return 2;
    }

    // Wait for the timer.

    if (WaitForSingleObject(hTimer, INFINITE) != WAIT_OBJECT_0)
        printf("WaitForSingleObject failed (%d)\n", GetLastError());
    else printf("Timer was signaled.\n");

    return 0;
}
```

Using Waitable Timers with an Asynchronous Procedure Call

The following example associates an asynchronous procedure call (APC) function, also known as a completion routine, with a waitable timer when the timer is set. The address of the completion routine is the fourth parameter to the **SetWaitableTimer** function. The fifth parameter is a void pointer that you can use to pass arguments to the completion routine.

The completion routine will be executed by the same thread that called **SetWaitableTimer**. This thread must be in an alertable state to execute the completion routine. It accomplishes this by calling the **SleepEx** function, which is an alertable function.

Each thread has an APC queue. If there is an entry in the thread's APC queue at the time that one of the alertable functions is called, the thread is not put to sleep. Instead, the entry is removed from the APC queue and the completion routine is called.

If no entry exists in the APC queue, the thread is suspended until the wait is satisfied. The wait can be satisfied by adding an entry to the APC queue, by a timeout, or by a handle becoming signaled. If the wait is satisfied by an entry in the APC queue, the thread is awakened and the completion routine is called. In this case, the return value of the function is **WAIT_IO_COMPLETION**.

After the completion routine is executed, the system checks for another entry in the APC queue to process. An alertable function will return only after all APC entries have been processed. Therefore, if entries are being added to the APC queue faster than they can be processed, it is possible that a call to an alertable function will never return. This is especially possible with waitable timers, if the period is shorter than the amount of time required to execute the completion routine.

When you are using a waitable timer with an APC, the thread that sets the timer should not wait on the handle of the timer. By doing so, you would cause the thread to wake up as a result of the timer becoming signaled rather than as the result of an entry being added to the APC queue. As a result, the thread is no longer in an alertable state and the completion routine is not called. In the following code, the call to **SleepEx** awakens the thread when an entry is added to the thread's APC queue after the timer is set to the signaled state.
C++Copy

```
#define UNICODE 1
#define _UNICODE 1

#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#define _SECOND 10000000

typedef struct _MYDATA {
   TCHAR *szText;
   DWORD dwValue;
} MYDATA;

VOID CALLBACK TimerAPCProc(
   LPVOID lpArg,               // Data value
   DWORD dwTimerLowValue,      // Timer low value
   DWORD dwTimerHighValue )    // Timer high value

{
   // Formal parameters not used in this example.
   UNREFERENCED_PARAMETER(dwTimerLowValue);
   UNREFERENCED_PARAMETER(dwTimerHighValue);

   MYDATA *pMyData = (MYDATA *)lpArg;

   _tprintf( TEXT("Message: %s\nValue: %d\n\n"), pMyData->szText,
          pMyData->dwValue );
   MessageBeep(0);

}

int main( void )
{
   HANDLE          hTimer;
   BOOL            bSuccess;
   __int64         qwDueTime;
   LARGE_INTEGER   liDueTime;
   MYDATA          MyData;

   MyData.szText = TEXT("This is my data");
   MyData.dwValue = 100;

   hTimer = CreateWaitableTimer(
           NULL,                   // Default security attributes
```

```
            FALSE,                  // Create auto-reset timer
            TEXT("MyTimer"));       // Name of waitable timer
    if (hTimer != NULL)
    {
       __try
       {
          // Create an integer that will be used to signal the timer
          // 5 seconds from now.
          qwDueTime = -5 * _SECOND;

          // Copy the relative time into a LARGE_INTEGER.
          liDueTime.LowPart  = (DWORD) ( qwDueTime & 0xFFFFFFFF );
          liDueTime.HighPart = (LONG)  ( qwDueTime >> 32 );

          bSuccess = SetWaitableTimer(
             hTimer,           // Handle to the timer object
             &liDueTime,       // When timer will become signaled
             2000,             // Periodic timer interval of 2 seconds
             TimerAPCProc,     // Completion routine
             &MyData,          // Argument to the completion routine
             FALSE );          // Do not restore a suspended system

          if ( bSuccess )
          {
             for ( ; MyData.dwValue < 1000; MyData.dwValue += 100 )
             {
                SleepEx(
                   INFINITE,     // Wait forever
                   TRUE );       // Put thread in an alertable state
             }

          }
          else
          {
             printf("SetWaitableTimer failed with error %d\n",
GetLastError());
          }

       }
       __finally
       {
          CloseHandle( hTimer );
       }
    }
    else
```

```
    {
        printf("CreateWaitableTimer failed with error %d\n", GetLastError());
    }

    return 0;
}
```

Using Critical Section Objects

The following example shows how a thread initializes, enters, and releases a critical section. It uses the **InitializeCriticalSectionAndSpinCount**, **EnterCriticalSection**, **LeaveCriticalSection**, and **DeleteCriticalSection** functions.

syntaxCopy

```
// Global variable
CRITICAL_SECTION CriticalSection;

int main( void )
{
    ...

    // Initialize the critical section one time only.
    if (!InitializeCriticalSectionAndSpinCount(&CriticalSection,
        0x00000400) )
        return;
    ...

    // Release resources used by the critical section object.
    DeleteCriticalSection(&CriticalSection);
}

DWORD WINAPI ThreadProc( LPVOID lpParameter )
{
    ...

    // Request ownership of the critical section.
    EnterCriticalSection(&CriticalSection);

    // Access the shared resource.

    // Release ownership of the critical section.
    LeaveCriticalSection(&CriticalSection);

    ...
return 1;
}
```

Using Condition Variables

The following code implements a producer/consumer queue. The queue is represented as a bounded circular buffer, and is protected by a critical section. The code uses two condition variables: one used by producers (`BufferNotFull`) and one used by consumers (`BufferNotEmpty`).

The code calls the **InitializeConditionVariable** function to create the condition variables. The consumer threads call the **SleepConditionVariableCS** function to wait for items to be added to the queue and the **WakeConditionVariable** function to signal the producer that it is ready for more items. The producer threads call **SleepConditionVariableCS** to wait for the consumer to remove items from the queue and **WakeConditionVariable** to signal the consumer that there are more items in the queue.

Windows Server 2003 and Windows XP: Condition variables are not supported.

C++Copy

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

#define BUFFER_SIZE 10
#define PRODUCER_SLEEP_TIME_MS 500
#define CONSUMER_SLEEP_TIME_MS 2000

LONG Buffer[BUFFER_SIZE];
LONG LastItemProduced;
ULONG QueueSize;
ULONG QueueStartOffset;

ULONG TotalItemsProduced;
ULONG TotalItemsConsumed;

CONDITION_VARIABLE BufferNotEmpty;
CONDITION_VARIABLE BufferNotFull;
CRITICAL_SECTION   BufferLock;

BOOL StopRequested;

DWORD WINAPI ProducerThreadProc (PVOID p)
{
    ULONG ProducerId = (ULONG)(ULONG_PTR)p;
```

```
    while (true)
    {
        // Produce a new item.

        Sleep (rand() % PRODUCER_SLEEP_TIME_MS);

        ULONG Item = InterlockedIncrement (&LastItemProduced);

        EnterCriticalSection (&BufferLock);

        while (QueueSize == BUFFER_SIZE && StopRequested == FALSE)
        {
            // Buffer is full - sleep so consumers can get items.
            SleepConditionVariableCS (&BufferNotFull, &BufferLock, INFINITE);
        }

        if (StopRequested == TRUE)
        {
            LeaveCriticalSection (&BufferLock);
            break;
        }

        // Insert the item at the end of the queue and increment size.

        Buffer[(QueueStartOffset + QueueSize) % BUFFER_SIZE] = Item;
        QueueSize++;
        TotalItemsProduced++;

        printf ("Producer %u: item %2d, queue size %2u\r\n", ProducerId,
Item, QueueSize);

        LeaveCriticalSection (&BufferLock);

        // If a consumer is waiting, wake it.

        WakeConditionVariable (&BufferNotEmpty);
    }

    printf ("Producer %u exiting\r\n", ProducerId);
    return 0;
}

DWORD WINAPI ConsumerThreadProc (PVOID p)
{
```

```
    ULONG ConsumerId = (ULONG)(ULONG_PTR)p;

    while (true)
    {
        EnterCriticalSection (&BufferLock);

        while (QueueSize == 0 && StopRequested == FALSE)
        {
            // Buffer is empty - sleep so producers can create items.
            SleepConditionVariableCS (&BufferNotEmpty, &BufferLock,
INFINITE);
        }

        if (StopRequested == TRUE && QueueSize == 0)
        {
            LeaveCriticalSection (&BufferLock);
            break;
        }

        // Consume the first available item.

        LONG Item = Buffer[QueueStartOffset];

        QueueSize--;
        QueueStartOffset++;
        TotalItemsConsumed++;

        if (QueueStartOffset == BUFFER_SIZE)
        {
            QueueStartOffset = 0;
        }

        printf ("Consumer %u: item %2d, queue size %2u\r\n",
            ConsumerId, Item, QueueSize);

        LeaveCriticalSection (&BufferLock);

        // If a producer is waiting, wake it.

        WakeConditionVariable (&BufferNotFull);

        // Simulate processing of the item.

        Sleep (rand() % CONSUMER_SLEEP_TIME_MS);
    }
```

```
    printf ("Consumer %u exiting\r\n", ConsumerId);
    return 0;
}

int main ( void )
{
    InitializeConditionVariable (&BufferNotEmpty);
    InitializeConditionVariable (&BufferNotFull);

    InitializeCriticalSection (&BufferLock);

    DWORD id;
    HANDLE hProducer1 = CreateThread (NULL, 0, ProducerThreadProc, (PVOID)1,
0, &id);
    HANDLE hConsumer1 = CreateThread (NULL, 0, ConsumerThreadProc, (PVOID)1,
0, &id);
    HANDLE hConsumer2 = CreateThread (NULL, 0, ConsumerThreadProc, (PVOID)2,
0, &id);

    puts ("Press enter to stop...");
    getchar();

    EnterCriticalSection (&BufferLock);
    StopRequested = TRUE;
    LeaveCriticalSection (&BufferLock);

    WakeAllConditionVariable (&BufferNotFull);
    WakeAllConditionVariable (&BufferNotEmpty);

    WaitForSingleObject (hProducer1, INFINITE);
    WaitForSingleObject (hConsumer1, INFINITE);
    WaitForSingleObject (hConsumer2, INFINITE);

    printf ("TotalItemsProduced: %u, TotalItemsConsumed: %u\r\n",
        TotalItemsProduced, TotalItemsConsumed);
    return 0;
}
```

Using One-Time Initialization

The following examples demonstrate the use of the one-time initialization functions.
Synchronous Example

In this example, the `g_InitOnce` global variable is the one-time initialization structure. It is initialized statically using **INIT_ONCE_STATIC_INIT**.

The `OpenEventHandleSync` function returns a handle to an event that is created only once. It calls the **InitOnceExecuteOnce** function to execute the initialization code contained in the `InitHandleFunction` callback function. If the callback function succeeds, `OpenEventHandleAsync` returns the event handle returned in *lpContext*; otherwise, it returns **INVALID_HANDLE_VALUE**.

The `InitHandleFunction` function is the one-time initialization callback function. `InitHandleFunction` calls the **CreateEvent** function to create the event and returns the event handle in the *lpContext* parameter.
C++Copy

```
#define _WIN32_WINNT 0x0600
#include <windows.h>

// Global variable for one-time initialization structure
INIT_ONCE g_InitOnce = INIT_ONCE_STATIC_INIT; // Static initialization

// Initialization callback function
BOOL CALLBACK InitHandleFunction (
    PINIT_ONCE InitOnce,
    PVOID Parameter,
    PVOID *lpContext);

// Returns a handle to an event object that is created only once
HANDLE OpenEventHandleSync()
{
  PVOID lpContext;
  BOOL  bStatus;

  // Execute the initialization callback function
  bStatus = InitOnceExecuteOnce(&g_InitOnce,          // One-time
initialization structure
                                InitHandleFunction,   // Pointer to
initialization callback function
```

```
                                NULL,                   // Optional parameter
to callback function (not used)
                                &lpContext);            // Receives pointer to
event object stored in g_InitOnce

  // InitOnceExecuteOnce function succeeded. Return event object.
  if (bStatus)
  {
    return (HANDLE)lpContext;
  }
  else
  {
    return (INVALID_HANDLE_VALUE);
  }
}

// Initialization callback function that creates the event object
BOOL CALLBACK InitHandleFunction (
    PINIT_ONCE InitOnce,        // Pointer to one-time initialization
structure
    PVOID Parameter,            // Optional parameter passed by
InitOnceExecuteOnce
    PVOID *lpContext)           // Receives pointer to event object
{
  HANDLE hEvent;

  // Create event object
  hEvent = CreateEvent(NULL,    // Default security descriptor
                       TRUE,    // Manual-reset event object
                       TRUE,    // Initial state of object is signaled
                       NULL);   // Object is unnamed

  // Event object creation failed.
  if (NULL == hEvent)
  {
    return FALSE;
  }
  // Event object creation succeeded.
  else
  {
    *lpContext = hEvent;
    return TRUE;
  }
}
```

Asynchronous Example

In this example, the `g_InitOnce` global variable is the one-time initialization structure. It is initialized statically using **INIT_ONCE_STATIC_INIT**.

The `OpenEventHandleAsync` function returns a handle to an event that is created only once. `OpenEventHandleAsync` calls the **InitOnceBeginInitialize**function to enter the initializing state.

If the call succeeds, the code checks the value of the *fPending* parameter to determine whether to create the event or simply return a handle to the event created by another thread. If *fPending* is **FALSE**, initialization has already completed so `OpenEventHandleAsync` returns the event handle returned in the *lpContext* parameter. Otherwise, it calls the **CreateEvent** function to create the event and the **InitOnceComplete** function to complete the initialization.

If the call to **InitOnceComplete** succeeds, `OpenEventHandleAsync` returns the new event handle. Otherwise, it closes the event handle and calls **InitOnceBeginInitialize** with **INIT_ONCE_CHECK_ONLY** to determine whether initialization failed or was completed by another thread.

If the initialization was completed by another thread, `OpenEventHandleAsync`returns the event handle returned in *lpContext*. Otherwise, it returns **INVALID_HANDLE_VALUE**.
C++Copy

```
#define _WIN32_WINNT 0x0600
#include <windows.h>

// Global variable for one-time initialization structure
INIT_ONCE g_InitOnce = INIT_ONCE_STATIC_INIT; // Static initialization

// Returns a handle to an event object that is created only once
HANDLE OpenEventHandleAsync()
{
  PVOID  lpContext;
  BOOL   fStatus;
  BOOL   fPending;
  HANDLE hEvent;

  // Begin one-time initialization
  fStatus = InitOnceBeginInitialize(&g_InitOnce,       // Pointer to one-time
initialization structure
```

```
                                    INIT_ONCE_ASYNC,    // Asynchronous one-
time initialization
                                    &fPending,          // Receives
initialization status
                                    &lpContext);        // Receives pointer to
data in g_InitOnce

  // InitOnceBeginInitialize function failed.
  if (!fStatus)
  {
    return (INVALID_HANDLE_VALUE);
  }

  // Initialization has already completed and lpContext contains event
object.
  if (!fPending)
  {
    return (HANDLE)lpContext;
  }

  // Create event object for one-time initialization.
  hEvent = CreateEvent(NULL,    // Default security descriptor
                       TRUE,    // Manual-reset event object
                       TRUE,    // Initial state of object is signaled
                       NULL);   // Object is unnamed

  // Event object creation failed.
  if (NULL == hEvent)
  {
    return (INVALID_HANDLE_VALUE);
  }

  // Complete one-time initialization.
  fStatus = InitOnceComplete(&g_InitOnce,             // Pointer to one-time
initialization structure
                             INIT_ONCE_ASYNC,         // Asynchronous
initialization
                             (PVOID)hEvent);          // Pointer to event
object to be stored in g_InitOnce

  // InitOnceComplete function succeeded. Return event object.
  if (fStatus)
  {
    return hEvent;
  }
```

```
  // Initialization has already completed. Free the local event.
  CloseHandle(hEvent);


  // Retrieve the final context data.
  fStatus = InitOnceBeginInitialize(&g_InitOnce,            // Pointer to
one-time initialization structure
                                    INIT_ONCE_CHECK_ONLY,   // Check whether
initialization is complete
                                    &fPending,              // Receives
initialization status
                                    &lpContext);            // Receives
pointer to event object in g_InitOnce

  // Initialization is complete. Return handle.
  if (fStatus && !fPending)
  {
    return (HANDLE)lpContext;
  }
  else
  {
    return INVALID_HANDLE_VALUE;
  }
}
```