Windows Hardware and Driver Central

Locks, Deadlocks, and Synchronization

April 5, 2006

Abstract

This paper explains how to use synchronization mechanisms to protect shared memory locations in kernel-mode drivers for the Microsoft® Windows® family of operating systems. By following the guidelines in this paper, driver writers will be able to determine when synchronization is required, what synchronization mechanisms are provided by the operating system, and how each type of synchronization mechanism is used.

This paper builds on topics that are introduced in the companion white paper "Scheduling, Thread Context, and IRQL" at http://www.microsoft.com/whdc/hwdev/driver/IRQL.mspx. Readers of this paper should be familiar with the information presented there.

Contents

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

Microsoft, Windows, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

1 Introduction

The Microsoft® Windows® family of operating systems provides a variety of mechanisms that drivers can use for synchronization. Synchronization is required for:

- Any shared data that multiple threads might access, unless all threads access it in a read-only manner.
- A set of operations that must be performed atomically.

To understand why synchronization is important, consider a situation in which two threads attempt to add one to the same global variable. This operation might require three processor instructions:

1. Read MyVar into a register.
2. Add 1 to the value in the register.
3. Write the value of the register into MyVar.

If the two threads run simultaneously on a multiprocessor system with no locks or other synchronization, the results of an update could be lost. For example, assume that the initial value of MyVar is 0 and that the operations proceed in the order shown below.

Threads Using No Locks on a Multiprocessor System

Thread A on Processor 1	Thread B on Processor 2
Read MyVar into register on Processor 1.	
	Read MyVar into register on Processor 2.
	Add 1 to Processor 2 register.
	Write Processor 2 register into MyVar.
Add 1 to Processor 1 register.	
Write Processor 1 register into MyVar.	

After both threads have updated MyVar, its value should be 2. However, the Thread B update is lost when Thread A increments the original value of MyVar and then overwrites the variable. Problems like this are often called *race conditions*.

On a single-processor system, *thread pre-emption* can also cause race conditions. At any time, the operating system may temporarily stop running a thread and instead run a different, higher-priority thread. When the system pre-empts a thread, the operating system saves the values of the processor's registers in the thread and restores them when the thread runs again.

The following example shows how a race condition can result from thread pre-emption. As in the previous example, assume that the initial value of MyVar is 0.

Threads Using No Locks on a Single Processor System

Thread A	Thread B
Read MyVar into register.	
Pre-empt Thread A; run Thread B.	
	Read MyVar into register. Add 1 to register. Write register into MyVar.
Pre-empt Thread B; run Thread A.	
Add 1 to register. Write register into MyVar.	

As in the multiprocessor example, the resulting value in MyVar is 1 instead of 2.

In both examples, using a lock to synchronize access to the variable would prevent the race condition. The lock ensures that Thread A has finished its update before Thread B accesses the variable, as shown in the following example.

Threads Using a Lock on Any System

Thread A	Thread B
Acquire lock.	
Read MyVar into register.	
Add 1 to register.	
Write register into MyVar.	
Release lock.	Acquire lock.
	Read MyVar into register.
	Add 1 to register.
	Write register into MyVar.
	Release lock.

The lock ensures that one thread's read and write operations are complete before another thread can access the variable. With locks in place, the final value of MyVar after these two code sequences complete is 2, which is the correct, intended result of the operation.

2 Choosing a Synchronization Mechanism

The best way to synchronize access in any particular situation depends on the operations that require synchronization. Synchronization methods fall into several broad categories. Within each category, Windows provides one or more specific mechanisms that drivers can use. Table 1 is a list of common synchronization methods.

Table 1. Common Synchronization Methods

Synchronization method	Description	Windows mechanisms
Interlocked operations	Provides atomic logical, arithmetic, and list manipulation operations that are both thread-safe and multiprocessor safe.	**Interlocked*Xxx*** and **ExInterlockedXxx** routines

Synchronization method	**Description**	**Windows mechanisms**
Mutexes	Provides (mutually) exclusive access to memory.	Spin locks, fast mutexes, kernel mutexes, synchronization events
Shared/exclusive lock	Allows one thread to write or many threads to read the protected data.	Executive resources
Counted semaphore	Allows a fixed number of acquisitions.	Semaphores

The following sections provide additional details about each of these synchronization methods.

2.1.1 Interlocked Operations

An interlocked operation completes a common task atomically. Windows provides interlocked routines to perform arithmetic and logical operations and to manipulate lists.

2.1.2 Mutexes

A mutex ensures mutually exclusive access; that is, while one thread has the mutex, all other threads are excluded from using it. A thread acquires the mutex before accessing the protected data and releases the mutex when access is complete. If the mutex is acquired, other threads must either spin or wait until it is released before they can acquire it.

Any lock that grants mutually exclusive access can be considered a mutex. For example, spin locks and synchronization events are both mutexes because only one thread can acquire a spin lock at a time and only one thread becomes eligible for execution when a synchronization event is signaled, The type of mutex that is appropriate for any particular situation depends on where and how the mutex will be used. When you are selecting a mutex, consider the following:

- At what IRQLs can the mutex be acquired and released?
- Does acquiring the mutex raise the current IRQL? If so, where is the old IRQL stored?
- Must the mutex be released on the same thread that acquired it?
- Can the mutex be acquired recursively? (That is, can a thread acquire the same mutex more than once without releasing it?)
- What happens to a mutex if the thread that is holding it is terminated? (This issue applies primarily to the mutexes that are used in user-mode applications.)

Table 2 is a list of the types of mutexes that are used in Windows drivers, along with their characteristics. Detailed information about each of the Windows mechanisms appears later in this paper.

Table 2. Windows Mutex Mechanisms

Type of mutex	IRQL considerations	Recursion and thread details
Interrupt spin lock	Acquisition raises IRQL to DIRQ and returns previous IRQL to caller.	Not recursive. Release on same thread as acquire.
Spin lock	Acquisition raises IRQL to DISPATCH_LEVEL and returns previous IRQL to caller.	Not recursive. Release on same thread as acquire.
Queued spin lock	Acquisition raises IRQL to DISPATCH_LEVEL and stores previous IRQL in lock owner handle.	Not recursive. Release on same thread as acquire.
Fast mutex	Acquisition raises IRQL to APC_LEVEL and stores previous IRQL in lock.	Not recursive. Release on same thread as acquire.
Kernel mutex (a kernel dispatcher object)	Enters critical region upon acquisition and leaves critical region upon release.	Recursive. Release on same thread as acquire.
Synchronization event (a kernel dispatcher object)	Acquisition does not change IRQL. Wait at IRQL <= APC_LEVEL and signal at IRQL <= DISPATCH_LEVEL.	Not recursive. Release on the same thread or on a different thread.
Unsafe fast mutex	Acquisition does not change IRQL. Acquire and release at IRQL <= APC_LEVEL.	Not recursive. Release on same thread as acquire.

2.1.3 Shared/Exclusive Locks

Shared/exclusive locks, also called read/write locks, allow one thread to have exclusive access to shared data or allow many threads to share access to the same data. The thread that has exclusive access can write the data; threads that share access can only read the data.

In Windows, executive resources provide a shared/exclusive lock.

2.1.4 Counted Semaphores

A counted semaphore is similar to a mutex, except that multiple threads can simultaneously acquire a semaphore. Counted semaphores are useful for protecting a set of identical data structures that are shared among several threads.

In Windows, the semaphore (a kernel dispatcher object) is a counted semaphore.

3 Windows Synchronization Mechanisms

Windows provides a variety of synchronization mechanisms, many of which are subject to IRQL restrictions. Therefore, you must consider the effects of thread interruption and understand the IRQLs at which your code might run.

When the operating system interrupts a thread, it forces that thread to temporarily run code at a higher interrupt request level (IRQL). (In other words, interruption is similar to a forced procedure call.)

Consider the case where code running at a low IRQL successfully acquires a lock, but the thread is interrupted to run code at a higher IRQL. If the higher-IRQL code tries to acquire the same lock, the thread may hang forever. The lower-IRQL code cannot run until the higher-IRQL code exits, but the higher-IRQL code cannot exit until the lower-IRQL code releases the lock. Only a single thread is involved. To prevent this problem, code that acquires a lock usually raises its IRQL to the highest IRQL at which any driver code that acquires the lock can run.

Table 3 is a list of the synchronization mechanisms available in Windows and the IRQL restrictions for each. Note that kernel dispatcher objects provide several types of synchronization that have a common structure and similar interfaces. The sections that follow this one describe each of the Windows mechanisms in detail.

Table 3. Summary of Windows Synchronization Mechanisms

Windows synchronization mechanism	Description	IRQL restrictions
Interlocked*Xxx* routines	Perform atomic logical and arithmetic operations on pageable data.	Call at any IRQL.
Spin locks	Allow exclusive access to data in nonpaged memory.	Acquire at IRQL <= DISPATCH_LEVEL.
ExInterlocked*Xxx* routines	Perform atomic logical, arithmetic, and list manipulation operations that are thread-safe and multiprocessor safe.	Call SList routines at IRQL <= DISPATCH_LEVEL; call other routines at any IRQL.
Fast mutexes	Protect data at APC_LEVEL, preventing thread suspension.	Acquire at IRQL <= APC_LEVEL.
Executive resources	Allow one thread to write or many threads to read the protected data.	Acquire at IRQL<=APC_LEVEL.
Kernel dispatcher objects (events, kernel mutexes, semaphores, timers, files, threads, processes)	Provide various types of synchronization at IRQL<=APC_LEVEL; can be used for synchronizing with user-mode applications.	Wait at IRQL <= APC_LEVEL; signal at IRQL <= DISPATCH_LEVEL.
Callback objects	Provides kernel-mode code synchronization at IRQL<=DISPATCH_LEVEL; can be used to synchronize activity between drivers.	Notify at IRQL <= DISPATCH_LEVEL; callback routines are called in the context of the notifying thread at the same IRQL at which notification occurred.

4 Interlocked*Xxx* Routines

The **Interlocked*Xxx*** routines perform common arithmetic and logical operations atomically, ensuring correct results on SMP systems. Whenever possible, drivers should use these routines. Most of them are native processor instructions and therefore do not require a lock.

The **Interlocked*Xxx*** routines can be used with pageable data. They are usually implemented inline by the compiler and can be called at any IRQL.

5 Spin Locks

A spin lock does exactly what its name implies: while one thread owns a spin lock, any other threads that are waiting to acquire the lock “spin” on a memory location in a busy wait until the lock is available. The threads do not block—that is, they are not suspended or paged out; they retain control of the CPU, thus preventing execution of other code at the same or a lower IRQL.

Spin locks are opaque objects of type KSPIN_LOCK. They must be allocated from nonpaged memory, such as the device extension of a driver-created device object or nonpaged pool allocated by the caller. Windows defines several types of spin locks, as described in Table 4.

Table 4. Windows Spin Locks

Type of spin lock	Description
Ordinary spin lock	Protects shared data at DISPATCH_LEVEL or higher. Used with **ExInterlocked*Xxx*** routines and elsewhere throughout drivers. (Drivers for Windows XP or later versions of Windows should use queued spin locks instead of ordinary spin locks.)
Queued spin lock	Protects shared data at DISPATCH_LEVEL or higher. Used with **ExInterlocked*Xxx*** routines and elsewhere throughout drivers. Queued spin locks are supported on Windows XP and later versions of Windows.
Interrupt spin lock	Protects shared data at DIRQL. Used in *InterruptService* and *SynchCritSection* routines.

All types of spin locks raise the IRQL to DISPATCH_LEVEL or higher. Spin locks are the only synchronization mechanism that can be used at IRQL >= DISPATCH_LEVEL. Code that holds a spin lock runs at IRQL >= DISPATCH_LEVEL, which means that the system’s thread switching code (the dispatcher) cannot run and, therefore, the current thread cannot be pre-empted. Therefore, drivers should hold spin locks for only the minimum required amount of time and eliminate from the locked code path any tasks that do not require locking. Holding a spin lock for an unnecessarily long duration can hurt performance system-wide.

All code within the spin lock must conform to the guidelines for running at IRQL >= DISPATCH_LEVEL. Every driver writer should understand these rules. For example, code within the spin lock must not cause a page fault because at IRQL >= DISPATCH_LEVEL, the operating system cannot wait for the kernel dispatcher event that is set internally when paging I/O completes. A page fault within a spin lock causes the system to crash with the bug check value IRQL_NOT_LESS_OR_EQUAL. Additional restrictions also apply. For complete information about these guidelines, see the companion white paper “Thread Context, Scheduling, and IRQL,” which is available at http://www.microsoft.com/whdc/hwdev/driver/IRQL.mspx.

To implement spin locks on a single-processor system, the operating system has only to raise the IRQL, which prevents pre-emption of the current thread. Because no other threads can run concurrently, raising the IRQL is adequate to protect any shared structures. (Note, however, that the checked build of the operating system uses spin locks, even on single-processor systems.) On an SMP system, the operating system raises the IRQL and then spins by testing and setting a variable using an interlocked instruction.

5.1 Ordinary Spin Locks

Ordinary spin locks work at DISPATCH_LEVEL. To create an ordinary spin lock, a driver allocates a KSPIN_LOCK structure in nonpaged memory and then calls **KeInitializeSpinLock** to initialize it. Code that runs at IRQL < DISPATCH_LEVEL must acquire and release the lock by calling **KeAcquireSpinLock** and **KeReleaseSpinLock**. These routines raise the IRQL before acquiring the lock and then lower the IRQL upon release of the lock.

Code that is already running at IRQL = DISPATCH_LEVEL should call **KeAcquireSpinLockAtDpcLevel** and **KeReleaseSpinLockFromDpcLevel** instead. These routines do not change the IRQL.

5.2 Queued Spin Locks

Queued spin locks are a more efficient variation of ordinary spin locks. Queued spin locks are available in Windows XP and later releases of Windows. Whenever multiple threads request the same queued spin lock, the waiting threads are queued in order of their request. In addition, queued spin locks test and set a variable that is local to the current CPU, so they generate less bus traffic and are more efficient on non-uniform memory architectures (NUMA).

A queued spin lock requires a KLOCK_QUEUE_HANDLE structure in addition to a KSPIN_LOCK structure. The KLOCK_QUEUE_HANDLE structure provides storage for a handle to the queue and the associated lock. This structure can be allocated on the stack. To initialize a queued spin lock, the driver calls **KeInitializeSpinLock**.

To ensure that the IRQL is properly raised and lowered, driver routines that run at PASSIVE_LEVEL or APC_LEVEL must call **KeAcquireInStackQueuedSpinLock** and **KeReleaseInStackQueuedSpinLock** to acquire and release these locks Driver routines that run at DISPATCH_LEVEL should call **KeAcquireInStackQueuedSpinLockAtDpcLevel** and **KeReleaseInStackQueuedSpinLockFromDpcLevel** instead. These routines do not raise and lower the IRQL.

5.3 Interrupt Spin Locks

An interrupt spin lock protects data such as device registers that a driver's *InterruptService* routine and *SynchCritSection* routine access at DIRQL. When a device driver connects its interrupt object, the operating system creates an interrupt spin lock associated with that interrupt object. The driver is not required to allocate storage for the spin lock or to initialize it.

When an interrupt occurs, the system raises the IRQL on the processor to DIRQL for the interrupting device, acquires the default interrupt spin lock associated with the interrupt object, and then calls the driver's *InterruptService* routine. While the *InterruptService* routine is running, the processor IRQL remains at DIRQL and the operating system holds the corresponding interrupt spin lock. When the *InterruptService* routine exits, the operating system releases the lock and lowers the IRQL (unless another interrupt is pending at that level).

The system also acquires the default interrupt spin lock when a driver calls **KeSynchronizeExecution** to run a *SynchCritSection* routine. The operating system raises the IRQL to DIRQL for the device, acquires the lock, and invokes the *SynchCritSection* routine. When the routine exits, the operating system releases the lock and lowers the IRQL. Other driver routines that share data with the *InterruptService* routine or *SynchCritSection* routine must call **KeAcquireInterruptSpinLock** to acquire this lock before they can access the shared data. **KeAcquireInterruptSpinLock** is available on Windows XP and later releases of Windows.

Some types of devices can generate multiple interrupts at different levels. Examples include devices that support PCI 3.0 MSI-X, which generate message-signaled interrupts (MSI), and a few older devices that interrupt at more than one IRQL. Drivers that support such devices must serialize access to data among two or more *InterruptService* routines.

In this case, the driver must create a spin lock to protect shared data at the highest DIRQL at which any interrupt may arrive. When the driver connects its interrupt objects, it passes a pointer to the driver-allocated KSPIN_LOCK structure, along with the highest DIRQL at which the device interrupts. The operating system associates the driver-created spin lock and the DIRQL with the interrupt object.

When the operating system calls the *InterruptService* routines, it raises the IRQL to the DIRQL specified with the interrupt object and acquires the driver-created spin lock. The system also uses this lock when it runs a *SynchCritSection* routine. Other driver routines that share data with the *InterruptService* or *SynchCritSection* routine must call **KeAcquireInterruptSpinLock** to acquire this lock before they can access the shared data.

Note

The next version of Windows, Microsoft Vista™ includes significant changes to the interrupt architecture to support message-signaled interrupts. Specifically, the **IoConnectInterrupt** routine is deprecated. You should use its replacement, **IoConnectInterruptEx** in new drivers, and older versions of drivers should be updated to use this new routine if possible. The Windows DDL provides this routine for use in Windows 2000 and later releases of Windows. For more information about these upcoming changes, see the white paper “Interrupt Architecture Enhancements in Microsoft Windows Vista,” which is available at http://www.microsoft.com/whdc/hwdev/bus/pci/MSI.mspx.

6 ExInterlocked*Xxx* Routines

The **ExInterlocked*Xxx*** routines perform arithmetic and list manipulation operations. All of these routines (except for **ExInterlockedAddLargeStatistic**) use a driver-allocated spin lock.

The **ExInterlocked*Xxx*** routines are coded in assembly language and usually disable interrupts at the processor; in effect, they run at IRQL = HIGH_LEVEL. To protect data on SMP systems, the operating system raises the IRQL and acquires the spin lock before performing the operation. When the routine completes, the operating system releases the lock and returns the IRQL to its original value. Like other routines that run at IRQL>= DISPATCH_LEVEL, **ExInterlocked*Xxx*** routines can operate only on data in nonpaged memory. Therefore, any parameter passed to one of these routines must be allocated on the kernel stack, from nonpaged pool, or in the device extension of the device object.

Use the **ExInterlocked*Xxx*** routines to perform arithmetic operations on a shared variable that a driver also accesses elsewhere, perhaps as part of a larger structure or in a longer sequence of tasks. For example, assume that a driver maintains a structure that contains status information about its device. The driver’s *DpcForIsr* routine accesses this structure, as do several routines that run at IRQL = PASSIVE_LEVEL. Therefore, the driver must protect the structure with a spin lock. To update several fields of the structure in a tight code path, the driver acquires the spin lock and assigns new values to the fields. To update the value of a single field, the driver uses an **ExInterlocked*Xxx*** routine, passing the spin lock that protects the structure.

In addition to routines for arithmetic operations, Windows includes **ExInterlocked*Xxx*** routines for managing three types of lists:

- Singly linked lists
- Doubly linked lists
- S-lists (sequenced, singly linked lists)

Drivers often maintain internal lists of IRPs, buffers, or other objects. If two or more threads have access to a list, the driver must protect the list while items are inserted and removed. For singly linked and doubly linked lists, the system provides both interlocked and non-interlocked versions of the manipulation routines, as shown in Table 5.

Table 5. List Manipulation Routines

Purpose	Noninterlocked routine	Interlocked routine
Insert entry at front of singly linked list.	PushEntryList	ExInterlockedPushEntryList
Remove entry from front of singly linked list.	PopEntryList	ExInterlockedPopEntryList
Insert entry at front of doubly linked list.	InsertHeadList	ExInterlockedInsertHeadList
Remove entry from front of doubly linked list.	RemoveHeadList	ExInterlockedRemoveHeadList
Insert entry at end of doubly linked list.	InsertTailList	ExInterlockedInsertTailList
Remove entry from end of doubly linked list.	RemoveTailList	None.
Initialize doubly linked list.	InitializeListHead	None.
Check whether list has entries.	IsListEmpty	None.
Remove entry from doubly linked list.	RemoveListEntry	None.
Initialize S-list.	None.	ExInitializeSListHead
Insert entry at front of S-list.	None.	ExInterlockedPushEntrySList
Remove entry from end of S-list.	None.	ExInterlockedPopEntrySList
Remove all entries from an S-list.	None.	ExInterlockedFlushSList

The **ExInterlocked*Xxx*List** routines use a driver-allocated spin lock. These routines can be called at any IRQL, provided that the driver always accesses a given list by using the **ExInterlocked*Xxx*List** routines.

The **ExInterlocked*Xxx*SList** routines manipulate S-lists. An S-list is a sequenced, interlocked, singly linked list that is both thread-safe and multiprocessor safe. Every S-list has an associated spin lock and a sequence number. The spin lock is used to protect the list while entries are inserted or deleted. The sequence number is incremented each time an entry is inserted or deleted. On some hardware architectures, using a sequence number enables these routines to avoid using a spin lock.

A driver must be running at IRQL <= DISPATCH_LEVEL to call the S-list routines. S-lists are useful for maintaining caches because a driver can simply and quickly remove the most recently used item from the list. Internally, Windows uses S-lists to implement lookaside lists.

7 Fast Mutexes

Fast mutexes, also called *executive mutexes*, enable a driver to protect a region for exclusive access. While a thread holds a fast mutex, no other thread can acquire the mutex. A fast mutex is an opaque structure of type FAST_MUTEX, which must be allocated from nonpaged memory.

Fast mutexes have low overhead and do not require the use of the system-wide dispatcher lock. As their name implies, fast mutexes are faster and more efficient than kernel mutexes.

Code paths that are protected by a fast mutex run at IRQL=APC_LEVEL, thus disabling delivery of all APCs and preventing the thread from suspension. Table 6 is a summary of the fast mutex acquisition routines.

Table 6. Fast Mutex Acquisition Routines

Routine	Description
ExAcquireFastMutex	Raises the IRQL to APC_LEVEL before acquiring the fast mutex. Blocks until the mutex is available.
ExAcquireFastMutexUnsafe	Acquires the mutex at the current IRQL. Blocks until the mutex is available.
ExTryToAcquireFastMutex	Raises the IRQL to APC_LEVEL before acquiring the fast mutex. Does not block if the mutex is not available.

ExAcquireFastMutex and **ExAcquireFastMutexUnsafe** cause the thread to block until the mutex is available. **ExTryToAcquireFastMutex** returns FALSE immediately if another thread has already acquired the mutex. Both **ExAcquireFastMutex** and **ExTryToAcquireFastMutex** raise the IRQL to APC_LEVEL before acquiring the fast mutex. Drivers should use **ExAcquireFastMutexUnsafe** (which does not raise the IRQL) only if either of the following is true:

- The thread is already running at APC_LEVEL.
- The thread acquires the mutex within a critical region that was previously entered by a call to **KeEnterCriticalRegion** or **FsRtlEnterFileSystem**.

In either of these situations, user-mode and normal kernel-mode APC delivery has already been disabled for the thread.

To use a fast mutex, a driver must:

1. Allocate a structure of type FAST_MUTEX from nonpaged pool.
2. Initialize the fast mutex by calling **ExInitializeFastMutex**.
3. Immediately before accessing the protected region, acquire the fast mutex by calling **ExAcquireFastMutex**, **ExAcquireFastMutexUnsafe**, or **ExTryToAcquireFastMutex**.
4. Perform the required operations on the protected data.

5. Release the fast mutex by calling **ExReleaseFastMutex** or **ExReleaseFastMutexUnsafe**.

Fast mutexes have the following limitations:

- Fast mutexes cannot be acquired recursively. Attempting to do so causes a deadlock.
- Driver code that holds a fast mutex runs at IRQL=APC_LEVEL. Therefore, such code cannot call routines that can be called only at IRQL=PASSIVE_LEVEL, such as **IoBuildDeviceIoControlRequest**.
- Fast mutexes are not kernel dispatcher objects. Therefore, a driver cannot use the **KeWaitForMultipleObjects** routine to wait for a fast mutex and a kernel dispatcher object simultaneously.

8 Kernel Dispatcher Objects

The operating system defines several types of kernel dispatcher objects, which provide various types of synchronization. Kernel dispatcher objects are relatively simple to use and provide locks that can be held at IRQL=PASSIVE_LEVEL. Table 7 is a summary of the kernel dispatcher object types.

Table 7. Kernel Dispatcher Object Types

Object type	Description	IRQL restrictions
Kernel mutex	Provides mutually exclusive access to data at PASSIVE_LEVEL or APC_LEVEL.	Wait at IRQL <=APC_LEVEL.
Event	Provides synchronization under driver-determined conditions; can be used to synchronize with user-mode applications.	Wait at IRQL<=APC_LEVEL; set at IRQL<=DISPATCH_LEVEL.
Semaphore	Protects a group of identical objects.	Wait at IRQL<=APC_LEVEL; set at IRQL<=DISPATCH_LEVEL.
Timer	Provides notification or synchronization at an absolute or relative time.	Wait at IRQL <=APC_LEVEL; set at IRQL <=DISPATCH_LEVEL.
Threads, processes, and files	Synchronizes with the creation or termination of a thread or process, or the completion of I/O to a file.	Wait at IRQL <=APC_LEVEL.

A driver can pass a kernel dispatcher object to the **KeWaitForSingleObject** and **KeWaitForMultipleObject** routines. Using these routines, a driver can wait with a specified time-out and can wait for one or more objects simultaneously.

8.1 Common Features

The operating system manages all kernel dispatcher objects in the kernel's dispatcher database (hence their name). To manipulate any of these objects, the system must raise the IRQL to DISPATCH_LEVEL and acquire the system-wide dispatcher lock, which protects the dispatcher database. However, the dispatcher lock is used frequently by many components, so sometimes the system must wait for it. Such waits can slow driver performance. For this reason, drivers should use executive resources or fast mutexes instead of kernel dispatcher objects whenever possible.

Kernel dispatcher objects are all based on the same common header (DISPATCHER_HEADER), but each type of object has its own object-type-specific initialization and release routines. Kernel dispatcher objects must be allocated from nonpaged pool or in the device extension of the device object.

A driver can refer to a kernel dispatcher object by using either a handle or a pointer. If a driver uses a handle to refer to a kernel dispatcher object that it created in an arbitrary thread context, that driver must set the OBJ_KERNEL_HANDLE attribute for the object. Setting this attribute protects system security by preventing user-mode threads from accessing the handle.

Drivers usually use kernel dispatcher objects to wait for the results of a synchronous I/O operation. Highest-level drivers that create and send IRPs to lower-level drivers wait in the context of the thread that issued the I/O request. Lower-level drivers sometimes must wait in an arbitrary thread context to synchronize execution among driver routines that run at IRQL PASSIVE_LEVEL or APC_LEVEL. For example, a driver might wait in its *DispatchPnP* routine for an event that is set when certain device operations are complete. As a general rule, however, drivers should avoid blocking any thread other than the thread that initiated the current I/O request.

Kernel dispatcher objects have two states: *signaled* and *not signaled*. Signaling indicates that the object is available for acquisition. Thus, an object in the signaled state is not owned (acquired) by any thread. An object in the not-signaled state is owned by one or more threads. The type of object determines the state to which the object is initially set. For example, kernel mutexes are set to the signaled state immediately upon initialization, but events must be explicitly signaled by a call to the **KeSetEvent** routine.

Although each type of object has its own type-specific initialization and release routines, drivers use **KeWaitForSingleObject** and **KeWaitForMultipleObjects** to acquire any kernel dispatcher object. Using **KeWaitForSingleObject**, a driver can wait for a single kernel dispatcher object. Using **KeWaitForMultipleObjects**, a driver can wait for more than one kernel dispatcher object; the objects need not be of the same type. Each of these routines takes as parameters:

- A pointer to the object(s) that are to be acquired.
- A reason for the wait. Drivers that are waiting on behalf of a user request in the context of a user thread should specify **UserRequest**; otherwise, drivers should specify **Executive**. The value of this field is informational only.
- A Boolean value (*Alertable*) that indicates whether the thread should be alertable while it is waiting. For drivers, this is usually FALSE.
- A wait mode (*WaitMode*), either **KernelMode** or **UserMode**. For drivers, this is usually **KernelMode**. If one or more of the objects is a mutex, this value *must* be **KernelMode**.
- An optional time-out value, which indicates how long the thread is to wait before it times out.

8.1.1 IRQL Restrictions

A thread can signal a kernel dispatcher object at IRQL <= DISPATCH_LEVEL, but it can wait for such an object only at IRQL <= APC_LEVEL. This IRQL restriction means that a driver cannot wait for an event or other dispatcher object in an *IoCompletion* routine, in a *StartIo* routine, or in any deferred procedure call (DPC) routine, because these routines can be called at DISPATCH_LEVEL. Highest-level drivers that are called in the context of the user thread that made the I/O request can wait for kernel dispatcher objects in their read and write dispatch routines.

Lower-level drivers should not wait in these routines; the read and write dispatch routines of some lower-level drivers, particularly in the storage and USB stacks, can be called at IRQL = DISPATCH_LEVEL.

However, a thread can acquire a kernel dispatcher object at DISPATCH_LEVEL if it does not wait for the object. To acquire a kernel dispatcher object without waiting, a thread sets the time-out value to zero in its call to **KeWaitForSingleObject** or **KeWaitForMultipleObjects**. A time-out value equal to zero means that the driver does not wait for the object to be signaled; instead, the call returns immediately with a status that indicates whether an object was available and acquired or is not currently available. Because no waiting is involved, such a call is valid at DISPATCH_LEVEL.

This feature is useful for testing whether an object has been signaled. For example, a DPC routine that must perform a task in synchronization with some other routine might check to see whether the object has been signaled. If so, the DPC can perform the task. If not, the DPC can do some other, unrelated task, or it can queue a work item to perform the original task. Because the work item runs in a thread at IRQL=PASSIVE_LEVEL, the work item can wait for a nonzero period.

8.1.2 Alerts and Wait Modes

The *Alertable* and *WaitMode* parameters to **KeWaitForSingleObject** and **KeWaitForMultipleObjects** determine how the system handles user-mode APCs while the thread is waiting. Table 8 is a summary of the effects of these parameters on APC delivery.

Table 8. Effects of the Alertable and WaitMode Parameters on APC Delivery

Value of *Alertable* and *WaitMode* parameters	**Special kernel-mode APC**		**Normal kernel-mode APC**		**User-mode APC**	
	Terminate wait?	Deliver and run APC?	Terminate wait?	Deliver and run APC?	Terminate wait?	Deliver and run APC?
Alertable = TRUE *WaitMode* = **UserMode**	No	If (A*), then Yes	No	If (B**), then Yes	Yes	Yes, after thread returns to user mode
Alertable = TRUE *WaitMode* = **KernelMode**	No	If (A), then Yes	No	If (B), then Yes	No	No
Alertable = FALSE *WaitMode* = **UserMode**	No	If (A), then Yes	No	If (B), then Yes	No	No (with exceptions, such as CTRL+C to terminate)
Alertable = FALSE *WaitMode* = **KernelMode**	No	If (A), then Yes	No	If (B), then Yes	No	No

*A: IRQL < APC_LEVEL.

**B: IRQL < APC_LEVEL, thread not already in an APC, and thread not in a critical region.

The system delivers most user-mode APCs when a thread unwinds from kernel mode back to user mode after an alertable wait. User-mode APCs do not interrupt user-mode code. After an application queues a user-mode APC to a thread, the application can cause the system to deliver the APCs by calling a wait function with the *Alertable* parameter set to TRUE. (The user-mode terminate APC is an exception. This APC is queued to terminate a thread, and the system delivers it whenever the thread unwinds back to user mode from kernel mode, not only after an alertable wait.)

If a driver calls either **KeWaitForSingleObject** or **KeWaitForMultipleObjects** with the *Alertable* parameter set to TRUE and the *WaitMode* parameter set to **UserMode**, the wait aborts with STATUS_USER_APC or STATUS_ALERTED whenever a user-mode APC (or an alert) is pending. When the thread returns to user mode, the system automatically delivers the user-mode APC. Drivers should not call either of the **KeWait*Xxx*** routines with *Alertable* set to TRUE and *WaitMode* set to **UserMode** unless the application has explicitly requested delivery of user-mode APCs during the wait.

If a driver calls **KeWaitForSingleObject** or **KeWaitForMultipleObjects** with *WaitMode* set to **UserMode**, but *Alertable* set to FALSE, the wait will abort with STATUS_USER_APC if the thread is being terminated. However, the driver must be waiting at IRQL = PASSIVE_LEVEL and must not be in a critical region.

The wait mode also determines whether the thread's kernel-mode stack can be paged out while waiting. If *WaitMode* is set to **UserMode**, the system pages out the kernel-mode stack while the thread is waiting. Waiting in **UserMode** is safe only if the waiting driver is the only driver on the stack. If one or more other drivers is on the stack, one of those drivers might try to update a stack variable, thereby causing a page fault. If that driver is running at IRQL=DISPATCH_LEVEL or higher, the page fault will cause the system to crash. Because PnP driver stacks often include filter drivers, PnP drivers rarely set *WaitMode* to **UserMode**.

Note
For more information about alertable waits, APCs, and the rules for operating at IRQL=DISPATCH_LEVEL, see the companion white paper "Scheduling, Thread Context, and IRQL" and the section "Do Waiting Threads Receive Alerts and APCs" under "Kernel-Mode Driver Architecture" in the Windows DDK.

8.2 Events

Drivers use events to synchronize activities between kernel-mode threads or between kernel-mode threads and user-mode applications. Both user-mode and kernel-mode code can create events. Windows also defines several standard event objects in the \\KernelObject object directory. Drivers can wait on these standard events.

An event is a synchronization object of type KEVENT, which must be allocated in nonpaged memory. Events can be either named or unnamed. Drivers usually use named events only to synchronize with external processes, such a user-mode application or another driver. Internally, drivers use unnamed events. Windows supports two types of events, *notification events* and *synchronization events*. The two types of event differ in the actions taken when they are signaled, as described later under "Notification Events" and "Synchronization Events."

To create and initialize an unnamed event of either type, a driver allocates an object of type KEVENT in nonpaged memory. The driver then calls **KeInitializeEvent** and specifies the type of event as a parameter in the call. To create and initialize a named event, the driver calls **IoCreateNotificationEvent** or **IoCreateSynchronizationEvent**.

To wait for an event, a driver calls **KeWaitForSingleObject** or **KeWaitForMultipleObjects**, as described previously under “Kernel Dispatcher Objects.”

To signal an event, a driver calls **KeSetEvent**, which has three parameters: a pointer to the event, a priority boost, and the Boolean *Wait*. Setting *Wait* to TRUE indicates that the thread intends to call **KeWait*Xxx*** immediately after **KeSetEvent** returns. This parameter provides an optimization in the cases where the driver intends to wait for another kernel dispatcher object immediately.

Normally, drivers call **KeSetEvent** with Wait set to FALSE. When *Wait* is FALSE, **KeSetEvent** raises the IRQL to DISPATCH_LEVEL, acquires the dispatcher lock, modifies the signaled-state of the event object, satisfies any outstanding waits, unlocks the dispatcher database, lowers the IRQL to its original value, and returns.

If the Wait parameter is TRUE, however, **KeSetEvent** does not release the dispatcher lock or lower the IRQL. This optimization can prevent unnecessary context switches because the caller thus signals the event and waits in one atomic operation. If a driver uses this feature, it must call **KeSetEvent** from IRQL < DISPATCH_LEVEL and in a non-arbitrary thread context.

A driver routine that operates in a producer/consumer scenario might use this feature. Such a driver usually works with two events in the following way. The driver routine that produces data signals the first event to indicate that it is ready to send data. It then immediately waits for the second event to be signaled by another thread. The second thread sets the second event to indicate it has received the data and is ready for more. Drivers should use this feature only in the context of the thread that requested the I/O operation; a driver should avoid blocking an unrelated thread.

8.2.1 Notification Events

A notification event wakes every waiting thread and remains in the signaled state until it is explicitly reset by a call to **KeResetEvent**. In the Win32® API, notification events are called *manual reset events*.

Drivers typically use notification events to wait for the completion of IRPs that they allocate and send. For example, a driver might send an I/O control code (IOCTL) to lower drivers in its device stack by calling **IoBuildDeviceIoControlRequest**. One of the parameters to this routine is a pointer to an event object. After the driver routine creates and sends the IRP, it waits on the event object. When the IRP is complete, the I/O Manager signals the event, thus satisfying the wait. The event remains signaled until a call to **KeResetEvent** returns it to the non-signaled state.

8.2.2 Synchronization Events

Synchronization events, also called *auto-resetting events*, wake a single thread and immediately return to the non-signaled state. Drivers use synchronization events less frequently than notification events.

A driver for a device that requires a long time to initialize might wait on a synchronization event in its *StartDevice* routine to ensure that its device is fully initialized. The driver’s *DpcForIsr* routine signals the event after the device has interrupted and any additional processing at IRQL DISPATCH_LEVEL is complete. Control then returns to the *StartDevice* routine, which can continue with device and driver initialization. Similarly, a driver might wait on a synchronization event in its *DispatchPnp* routine to ensure that I/O has completed before stopping or removing the device.

8.2.3 Synchronizing with User-Mode Applications

Kernel-mode drivers cannot make calls to user-mode routines. However, a kernel-mode driver with a closely-coupled application might sometimes need to notify the application about a device- or driver-related occurrence. There are several ways to implement such notification; two recommended methods are outlined in this section.

One way to coordinate activities between a kernel-mode driver and a user-mode application is to share an event. In the driver:

1. Define a private I/O control code (IOCTL) with which a user-mode application can pass an event.
2. Provide a *DispatchDeviceControl* routine that handles the private IOCTL supplied in IRP_MJ_DEVICE_CONTROL requests.
3. Validate the handle received in the IOCTL by calling **ObReferenceObjectByHandle**. In the *DesiredAccess* parameter, specify SYNCHRONIZE access, and in the *ObjectType* parameter, specify ***ExEventObjectType**.
6. To signal the event, call **KeSetEvent**; to reset a notification event, call **KeResetEvent**.
7. Call **ObDereferenceObject** to free the handle when the event is no longer needed.

In the user-mode application:

1. Create a named event by calling **CreateEvent**.
2. Pass the handle to the event to the driver by calling **DeviceIoControl**, specifying the driver-defined IOCTL.
3. To wait for the kernel-mode driver to signal the event, call **WaitForSingleObject** or **WaitForMultipleObjects**.
4. Delete the event object before exiting by calling **CloseHandle**.

This technique is suitable in situations where the driver routine and the user-mode thread that share the event always run in the same process context. However, in the layered WDM driver model, lower-level driver routines are not usually called in the context of the requesting thread or process.

A more general approach that eliminates thread context problems is to use a **DeviceIoControl** request without an event. In this technique, the driver defines a private IOCTL for the I/O request. The application creates a dedicated thread that sends the **DeviceIoControl** request to the driver, which returns STATUS_PENDING. To notify the user-mode application, the driver completes the request. Because this technique does not depend upon the validity of user-mode data, it is suitable for use in lower-level drivers.

For an example of both of these techniques, see the event sample (src\general\event) in the Windows DDK.

8.3 Kernel Mutexes

Kernel mutexes, often just called *mutexes*, are useful for synchronizing access to memory in pageable code or over a relatively long period of time. A kernel mutex ensures that a thread has exclusive access to the protected data. Drivers can use kernel mutexes at IRQL <= APC_LEVEL.

Kernel mutexes depend on thread context. Driver routines that use kernel mutexes are usually highest-level driver routines that run in the context of the thread that requested the I/O operation. A driver routine that acquires a mutex should release it within the same thread context.

Kernel mutexes differ from fast mutexes in the following ways:

- Kernel mutexes can be acquired recursively; fast mutexes cannot.
- Kernel mutexes are acquired by using **KeWaitForSingleObject**, **KeWaitForMultipleObjects**, and **KeWaitForMutexObject**. Fast mutexes are acquired by using **ExAcquireFastMutex**, **ExTryToAcquireFastMutex**, and **ExAcquireFastMutexUnsafe**.
- Kernel mutexes require the use of the system-wide dispatcher lock. Therefore, they have greater overhead and are less efficient than fast mutexes.

To use a kernel mutex, a driver must:

1. Allocate a KMUTEX data structure in nonpaged memory such as the device extension of a driver-created device object or nonpaged pool.
2. Initialize the mutex by calling **KeInitializeMutex**, passing a pointer to the previously allocated data structure. (The *Level* parameter is ignored.)
3. Wait for the mutex by calling **KeWaitForSingleObject**, **KeWaitForMultipleObjects**, or **KeWaitForMutexObject**.
4. Perform the required operations on the protected data.
5. Release the mutex by calling **KeReleaseMutex**.

The operating system initializes every kernel mutex to the signaled state. Consequently, the first thread's initial call to wait for the mutex succeeds immediately and returns.

Driver routines should always specify **KernelMode** when they are waiting for a kernel mutex. Waiting in kernel mode prevents the thread's kernel-mode stack from being paged out and disables the delivery of user-mode and normal kernel-mode APCs, thus preventing thread termination and suspension. Special kernel-mode APCs, such as the special kernel-mode APC for I/O completion, can still be delivered. Internally, acquiring a kernel mutex calls **KeEnterCriticalRegion**. If the thread is running at PASSIVE_LEVEL when it acquires the mutex, this call disables the delivery of normal kernel-mode APCs until the thread releases the mutex. If the thread is running at APC_LEVEL when it acquires the mutex, entering a critical region has no effect because normal kernel-mode APC delivery is already disabled.

A thread that holds a mutex must release the mutex before any transition to user mode; the system crashes if a thread holds a mutex during the transition. For example, a highest-level driver that acquires a mutex while servicing a user-mode I/O request must release the mutex before returning to the user-mode code.

When a thread releases a mutex, it passes a *Wait* parameter. The *Wait* parameter has the same meaning in **KeReleaseMutex** as in **KeSetEvent**; see the section "Events" for details.

A thread that acquires a mutex recursively must release the mutex the same number of times as it acquired the mutex. The operating system does not signal the mutex or call **KeLeaveCriticalRegion** until all acquisitions have been released.

8.4 Semaphores

A semaphore is similar to a mutex, except that multiple threads can simultaneously acquire a semaphore. Semaphores are useful for protecting a set of identical data structures that are shared among several threads.

Every semaphore has a limit and a count. The *limit* is the maximum number of threads that can acquire the semaphore at a time, and the *count* is the number of threads that can currently acquire the semaphore.

For example, a driver might allocate several buffers for I/O and protect them with a semaphore. The semaphore's limit is the number of buffers. When a driver routine needs an I/O buffer, it waits for the semaphore. If the semaphore's count is equal to zero, all buffers are in use. If the count is equal to its limit, all buffers are free.

To use a semaphore, a driver must:

1. Allocate a KSEMAPHORE data structure in the device extension of a driver-created device object or in nonpaged pool allocated by the caller.
2. Initialize the semaphore by calling **KeInitializeSemaphore**, specifying the semaphore's count and limit. Setting the count to 0 initializes the semaphore in the not-signaled state; setting the count greater than 0 signals the semaphore and indicates how many threads can acquire it initially.
3. Wait for the semaphore by calling **KeWaitForSingleObject** or **KeWaitForMultipleObjects**.
4. Perform the required operations on the protected data.
5. Release the semaphore by calling **KeReleaseSemaphore**, passing a value to add to the current count.

If a driver increases the count of a semaphore above the semaphore's limit, the system raises an exception. Such an error could occur if the driver attempts to release the semaphore too many times. This behavior is different from that of events; setting an already signaled event has no effect.

When a thread releases a semaphore, it can also specify a *Wait* parameter. The *Wait* parameter has the same effect for a semaphore as for a mutex or for an event; see the section "Events" for details.

A thread can determine whether a semaphore is signaled or not signaled by calling **KeReadStateSemaphore**.

8.5 Timers

Drivers often use timers for polling and handling device time-outs. Like events, timers can be used for synchronization or for notification. A driver creates a notification timer by calling **KeInitializeTimer**; it can create either a notification timer or a synchronization timer by calling **KeInitializeTimerEx**.

Both types of timers expire after a specified interval at either an absolute or relative time. Absolute times are measured in 100-nanosecond units starting on January 1, 1601. This time is tied to the calendar—advancing the clock by one hour brings the expiration time one hour closer. Relative times are negative numbers measured in 100-nanosecond units from the moment the timer was started. Relative time is measured in machine running time and is unaffected by system clock changes. Relative time includes time spent sleeping. When the computer awakens, the operating system adjusts the internal machine time to include the time that the computer was asleep. As a result, many timers expire simultaneously as soon as the operating system resumes. When a notification timer expires, all waiting threads are released. The timer remains in the signaled state until a thread explicitly resets by calling **KeSetTimer**. When a synchronization timer expires, a single waiting thread is released and the operating system immediately resets the timer to the non-signaled state.

A driver can wait on a timer object at IRQL <= APC_LEVEL, or it can specify a *CustomTimerDpc* routine to be called when the timer expires. Drivers can use *CustomTimerDpc* routines instead of driver-created threads to perform short-lived operations. *CustomTimerDpc* routines are also used to time out a request at IRQL=DISPATCH_LEVEL.

To use a timer, a driver should do the following, as necessary:

1. Allocate a structure of type KTIMER in nonpaged memory.
2. Create and initialize the timer by calling **KeInitializeTimer** or **KeInitializeTimerEx**. **KeInitializeTimer** creates a notification timer. **KeInitializeTimerEx** creates a notification timer or a synchronization timer.
3. To associate the timer with a *CustomTimerDpc* routine, call **KeInitializeDpc** to initialize a DPC object and register the *CustomTimerDpc* routine.
4. Set the timer by calling **KeSetTimer** or **KeSetTimerEx**, specifying the interval at which the timer expires. To queue the *CustomTimerDpc* routine when the timer expires, include the optional *Dpc* parameter.
5. To wait on a timer object, call **KeWaitForSingleObject** or **KeWaitForMultipleObjects***.*
6. To cancel a timer before it expires, call **KeCancelTimer**.
7. To reset a notification timer after it expires, call **KeSetTimer**.

Both notification and synchronization timers can be recurring (or *periodic*) timers. As soon as a periodic timer's interval expires, the operating system immediately queues the timer again. Consequently, a DPC routine that is associated with a periodic timer can run simultaneously on more than one CPU in an SMP system. Such simultaneous execution can occur, for example, if the DPC routine takes longer to run than the timer interval or if its execution is delayed because other DPCs precede it in the DPC queue. Because the DPC routines run at IRQL = DISPATCH_LEVEL, drivers must use spin locks to protect any data that these routines share.

If your driver uses more than one timer object, DPC object, or *CustomTimerDpc* routine, you should understand the order in which the operating system signals the objects and queues the DPCs, the consequences of using these objects in varying combinations, and the effects of canceling one or more of the timers. See "Using a CustomTimerDpc Routine" in the "Kernel Mode Drivers Architecture Design Guide" section of the Windows DDK for details.

8.6 Threads, Processes, and Files

Threads, processes, and files are also kernel dispatcher objects. Drivers can use the **KeWait*Xxx*** routines to synchronize actions with the termination of a thread or process or with the completion of I/O to a file. In addition, a driver can request notification when a new thread or process is created.

To synchronize with a particular process, thread, or file, the driver must get a pointer to the object that represents that process, thread, or file. A driver that creates a thread can pass the handle returned by the **PsCreateSystemThread** routine to **ObReferenceObjectByHandle** to get a pointer to the thread object. Similarly, a driver can get a pointer to a file object by passing a file handle to **ObReferenceObjectByHandle**. The resulting object pointers can then be passed to **KeWait*Xxx***.

To wait on a thread, process, or file object, a kernel-mode driver must specify a **KernelMode** wait. Waiting in kernel mode prevents paging of the waiting thread's stack and disables user-mode and normal kernel-mode APCs. The wait is satisfied when the thread or process terminates or when the current file I/O operation is complete.

A file I/O operation (a single IRP) is complete when the operating system signals an internal event that is embedded in the file object. Every file object has such an embedded event. The event is a synchronization (auto-reset) event; that is, the event is reset as soon as a waiting thread is notified. Applications and file system drivers that implement asynchronous I/O can wait on the file event to find out when the I/O operation completes.

Synchronizing with a specific thread is generally useful only in drivers that create device-dedicated or other driver-specific threads. Most driver routines, except for the I/O dispatch routines of highest-level drivers, are called in the context of an arbitrary thread. Consequently, synchronizing driver activity with the current thread context is rarely meaningful.

A driver can also request notification whenever a thread or process is created or deleted system-wide. To do so, the driver sets a callback routine by calling **PsSetCreateProcessNotifyRoutine** or **PsSetCreateThreadNotifyRoutine**. The operating system calls the routine any time a process or thread is created or deleted. A driver that establishes thread or process callback routines must not exit before the operating system shuts down.

9 Executive Resources

By using an executive resource, a driver can implement a read/write lock. Executive resources are designed for use with data structures that require exclusive access for writing but that can be read by several threads concurrently. Executive resources are not maintained in the system's dispatcher database, so they usually are faster and more efficient than kernel dispatcher objects. A thread can acquire an executive resource for exclusive (write) access or for shared (read) access. Code that runs at IRQL=PASSIVE_LEVEL or APC_LEVEL can use executive resources.

An executive resource is a structure of type ERESOURCE, which must be allocated in nonpaged memory (for example, the device extension of the device object or nonpaged pool). An ERESOURCE structure must be naturally aligned; that is, the structure must be aligned on a 4-byte boundary on a 32-bit system and on an 8-byte boundary on a 64-bit system. The ERESOURCE structure itself is opaque to driver writers.

Table 9 is a summary of the acquisition routines for executive resources. It includes the type of access that each routine provides and when such access is granted.

Table 9. Executive Resource Acquisition Routines

Routine	Type of access	Conditions
ExAcquireResourceSharedLite	Shared	Acquires resource if either: The resource is not already acquired exclusively and no thread is waiting to acquire exclusive access. or The requesting thread already has shared or exclusive access.
ExAcquireResourceExclusiveLite	Exclusive	Acquires resource if the resource is not already acquired for shared or exclusive access.

Routine	Type of access	Conditions
ExAcquireSharedStarveExclusive	Shared	Acquires resource if either: The resource is not already acquired exclusively. or The requesting thread already has shared or exclusive access. Threads waiting for exclusive access continue to wait.
ExAcquireSharedWaitForExclusive	Shared	Same as ExAcquireResourceSharedLite except that: If the requesting thread already has access to the resource and one or more threads are waiting for exclusive access, the recursive request blocks until the exclusive requests have been satisfied.
	Exclusive	Same as **ExAcquireResourceExclusiveLite**, but it does not block if access is not available.

If a thread acquires a resource for exclusive access, it can later convert to shared access. However, a thread cannot convert shared access to exclusive access. The **ExConvertExclusiveToSharedLite** routine changes a thread's access from Exclusive to Shared and grants shared access to any additional threads that are waiting for shared access. On a checked build, the system ASSERTs if the requesting thread does not own exclusive access to the resource.

One thread can release a resource on behalf of another by calling the routine **ExReleaseResourceForThread**. File system drivers use this routine when one thread acquires a resource, partially processes an I/O request, and then posts the I/O request to another thread. In this case, the thread that completes the I/O request can call this routine to release the resource on behalf of the first thread.

A driver can determine whether a resource has already been acquired by any thread using the utility routines **ExIsResourceAcquiredLite**, **ExIsResourceAcquiredSharedLite**, and **ExIsResourceAcquiredExclusiveLite**. In addition, a driver can determine how many other threads are waiting for either shared or exclusive access to a resource by calling **ExGetSharedWaiterCount** or **ExGetExclusiveWaiterCount**.

Suspending a thread that owns an executive resource can cause a deadlock. For example, assume that Thread 1 has shared access to a resource and that Thread 2 is waiting for exclusive access to the same resource. If Thread 1 is suspended, Thread 2 could wait forever for the resource. A malicious user could intentionally create a deadlock in this manner to mount a denial-of-service attack on the driver and, perhaps, on the entire operating system. For this reason, drivers must prevent thread suspension while holding an executive resource. For details, see "Security Issues" later in this paper.

10 To use an executive resource, a driver must:

1. Allocate an ERESOURCE structure from nonpaged pool.

2. Initialize the resource by calling **ExInitializeResourceLite**, usually from a **DriverEntry** or **AddDevice** routine.
3. Disable normal kernel-mode APCs before acquiring the resource. A device driver calls **KeEnterCriticalRegion**; a file system driver calls **FsRtlEnterFileSystem**. If the driver routine is running in the context of a system thread, however, it generally does not need to disable APCs because the thread is unlikely to be suspended.
4. Acquire the resource by calling one of the resource acquisition routines listed in Table 9.
5. Perform the required operations on the protected data.
6. Release the resource by calling **ExReleaseResourceLite**.
7. Re-enable normal kernel-mode APCs by calling **KeLeaveCriticalRegion** or **FsRtlLeaveFileSystem**.

All of the resource acquisition routines return a Boolean value that indicates whether acquisition succeeded. When the thread acquires the resource, the acquisition routine returns TRUE. If the driver does not block and the resource is not available, the routine returns FALSE.

Unlike fast mutexes and spin locks, executive resources can be acquired recursively. A thread that acquires an executive resource recursively must release the resource as many times as it was acquired. Recursive acquisition of resources is common in file system drivers. For example, a file system driver might implement a cache by mapping files into a reserved area in virtual memory. The driver holds certain locks while it processes the cached data. If that processing causes a page fault, the operating system generates an additional I/O request, which is sent to the same file system driver and interrupts that driver's processing of the cached I/O. To handle the additional I/O request, the file system driver must recursively acquire some of the same locks it uses when processing the cached I/O.

Executive resources are similar to fast mutexes in that a thread that tries to acquire a resource while another thread has exclusive access to it will block. While this thread is waiting, other threads in the Ready state run.

11 Callback Objects

Callback objects are useful for synchronization and notification between kernel-mode routines. Callback objects are kernel-mode only; they cannot be shared with user-mode applications.

A driver creates a callback object by calling **ExCreateCallback**. Users of the object register a callback routine by calling **ExRegisterCallback**. When the driver-specified callback conditions occur, the driver calls **ExNotifyCallback** to request that the callback routines be run. **ExNotifyCallback** can be called at IRQL<=DISPATCH_LEVEL, and the callback routines are called at the same IRQL at which **ExNotifyCallback** was called, in the context of the notifying thread. If your driver registers a callback routine, be sure that you know the IRQL at which notification takes place and code the callback routine appropriately.

12 Driver-Defined Locks

In addition to the synchronization mechanisms provided by the operating system, drivers can define their own locks. If you implement a driver-defined lock, you must keep in mind that optimizing compilers and certain hardware architectures sometimes reorder read and write instructions to improve performance. To prevent such reordering, driver code sometimes requires a memory barrier.

A *memory barrier* is a processor instruction that preserves the ordering of read and write operations, as seen from the perspective of any other processor. The operating system's locking mechanisms (spin locks, fast mutexes, kernel dispatcher objects, and executive resources) all have implied memory barriers that preserve the ordering of instructions.

If you create your own locks, you might need to put memory barriers in the locked code to ensure the correct results. The **ExInterlocked*Xxx*** and **Interlocked*Xxx*** routines and the **KeMemoryBarrier** and **KeMemoryBarrierWithoutFence** routines insert memory barriers to prevent such reordering.

For details about memory barriers and processor reordering, see the white paper "Memory Barriers on Multiprocessor Architectures," which is available at www.microsoft.com/hwdev.

13 Using Multiple Synchronization Mechanisms Simultaneously

Attempting to acquire two or more synchronization mechanisms at once can cause a deadlock if this is done improperly. For this reason, the Windows DDK advises driver writers to never acquire more than one lock at a time. However, in some situations, using multiple locks is appropriate, or even necessary.

For example, a driver might maintain two lists that require protection at IRQL DISPATCH_LEVEL. Most code accesses only one of the lists at any given time. Occasionally, however, a driver routine must move an item from one list to another. Using a single spin lock to protect both lists is inefficient. If a driver routine running in Thread 1 acquires the lock to update the first list, another driver routine running in Thread 2 must wait to access the second list.

A better solution is to protect each list with its own spin lock. Code that accesses List A acquires the spin lock for List A. Code that accesses List B acquires the spin lock for List B. Code that accesses both lists acquires both locks. To prevent deadlocks, code that acquires both locks must always acquire the locks in the same order.

To determine the proper order for acquiring the locks, you should establish a *lock hierarchy* for your code. The hierarchy ranks the locks in order of increasing IRQL. List the lock that requires the lowest IRQL first, the second lowest IRQL next, and so forth. When driver code must acquire multiple locks at once, it should acquire them in order of increasing IRQL. A code sequence that requires more than one lock at the same IRQL should acquire the most frequently used lock first.

Similarly, a code sequence that uses multiple locks should release them in the inverse of the order in which it acquired them. That is, it should release the most recently acquired lock first.

If the driver follows the locking hierarchy consistently, deadlocks will not occur. However, if the driver violates the hierarchy, deadlocks are inevitable.

14 Preventing Deadlocks

A deadlock occurs when a thread waits for something it can never acquire. For example, a thread that holds a spin lock cannot recursively acquire the same spin lock. The thread will spin forever waiting for itself to release the lock.

Two threads can create a mutual deadlock (sometimes called a *deadly embrace*) if each holds a lock that the other is trying to acquire. For example, assume a driver has created spin locks to protect two structures, A and B. Thread 1 acquires the lock that protects Structure A and Thread 2 acquires the lock that protects Structure B. If Thread 1 now attempts to acquire the lock for B, and Thread 2 attempts to acquire the lock for A, the threads deadlock. Neither can acquire the second lock until the other thread releases it. Establishing and following lock hierarchies prevents deadly embraces.

Follow these guidelines to prevent deadlocks:

- Never wait on a kernel dispatcher object in any driver routine that can be called at IRQL>=DISPATCH_LEVEL. Routines that can be called at IRQL>=DISPATCH_LEVEL include *IoCompletion* routines and the I/O dispatch routines of storage drivers and USB hub drivers. If in doubt, use the ASSERT() macro on a checked build to test for the IRQL at which the routine is called.
- Disable normal kernel-mode APC delivery before calling any of the executive resource acquisition routines and before calling **KeWait*Xxx*** to wait on an event, semaphore, timer, thread, file object, or process. In a device driver, call **KeEnterCriticalRegion** and **KeLeaveCriticalRegion** to disable and subsequently re-enable APC delivery. In a file system driver, call **FsRtlEnterFileSystem** and **FsRtlLeaveFileSystem**.
- Use the Driver Verifier (verifier.exe) Deadlock Detection option to find potential deadlocks. This option is available on Windows XP and later releases of Windows.
- Always establish and follow a lock hierarchy in code that acquires more than one lock at any given time.

15 Security Issues

Drivers that use locks at IRQL PASSIVE_LEVEL outside a critical region are open to denial of service attacks if the thread that holds the lock is suspended. This problem occurs because Windows queues a normal kernel-mode APC to suspend the thread. Even if the driver specifies a **KernelMode** wait, normal kernel-mode APCs are delivered whenever all of the following are true:

- The target thread is running at IRQL < APC_LEVEL.
- The target thread is not already running an APC.
- The target thread is not in a critical region; that is, it did not call **KeEnterCriticalRegion** before calling **KeWait*Xxx***.

Note
Kernel mutexes and fast mutexes do not have this problem. The operating system enters a critical region before it acquires a fast mutex or kernel mutex on behalf of a thread.

Consider the following scenario:

While it is handling an I/O request from a user-mode application, a driver waits for a kernel dispatcher object (other than a mutex) or acquires an executive resource. The driver requests a **KernelMode** wait that is not alertable: that is, the *WaitMode* parameter is **KernelMode** and the *Alertable* parameter is FALSE.

Assume that the requesting application has a second thread running. If the second thread acquires a handle to the thread that requested the I/O operation, it can suspend the requesting thread, thus rendering the driver – and possibly the whole system—unusable.

To eliminate this possible security threat, a driver should enter a critical region (disable APCs) before calling **KeWait*Xxx***. To disable and subsequently re-enable APCs, a device driver calls **KeEnterCriticalRegion** and **KeLeaveCriticalRegion**; a file system driver calls the **FsRtlEnterFileSystem** and **FsRtlLeaveFileSystem** macros. A driver can check whether normal kernel-mode APCs are disabled by calling **KeAreApcsDisabled**. **KeAreApcsDisabled** is available on Windows XP and later releases of Windows.

16 Performance Issues

Although nearly every driver requires spin locks and other synchronization mechanisms, these mechanisms by their very nature can cause performance bottlenecks. Follow these guidelines to improve performance in synchronization code:

- Use locks only when necessary. For example, to gather statistical information, use per-processor data structures instead of a single, system-wide data structure that requires a lock.
- Use executive resources (ERESOURCE structures) to protect read/write data at IRQL < DISPATCH_LEVEL, so that multiple readers can be active at once.
- Use fast mutexes or executive resources instead of kernel dispatcher objects whenever possible. Because fast mutexes and executive resources are not maintained in the dispatcher database, the system can acquire them without using the dispatcher lock. As a result, they are faster and contribute to better performance system-wide.
- Use the **Interlocked*Xxx*** routines whenever possible. These routines do not acquire a spin lock and are therefore relatively fast.
- Use an in-stack queued spin lock instead of an ordinary spin lock whenever several components might frequently contend for the lock
- Use the **ExTryToAcquire*Xxx*** routines to acquire a lock whenever possible, particularly if you have established a locking hierarchy and are using multiple, nested locks. If you need to wait for the second or third lock in a hierarchy, consider releasing the locks that you have already acquired so that another thread that might need fewer locks can proceed, and then later reacquire the locks from the top of the hierarchy.
- Minimize the number of times your driver calls routines that use the dispatcher lock. Such routines include those to wait on a dispatcher object (**KeWaitForSingleObject**, **KeWaitForMultipleObjects**, and **KeWaitForMutexObject**) and calls that set and release dispatcher objects (**KeSetEvent**, **KeReleaseSemaphore**, and so forth). Frequent use of the dispatcher lock can slow performance system-wide because calls that require it must sometimes spin.
- Hold each spin lock for the minimum amount of time necessary, particularly if the lock is frequently required by other code. For example, traversing a long, linked list in a linear order while holding a heavily used spin lock can cause a performance bottleneck.

17 Best Practices for Driver Synchronization

To avoid problems related to synchronization in drivers, adopt these practices:

- Determine the highest IRQL at which any code can access the data.
- If any code that accesses the data runs at IRQL>=DISPATCH_LEVEL, you must use a spin lock.
- If all code runs at IRQL PASSIVE_LEVEL or APC_LEVEL, you can use an executive resource, a fast mutex, or one of the kernel dispatcher objects—whichever is best suited to the driver’s requirements.

- To synchronize driver execution with a user-mode application, define a private IOCTL and use either an event that is defined by the user-mode application or an I/O request that the driver completes to notify the application.
- To prevent thread suspension, a driver should enter a critical region before acquiring an executive resource or waiting for a kernel-dispatcher object (other than a mutex) at IRQL=PASSIVE_LEVEL.
- To manage lists or to perform arithmetic or logical operations on a single memory location, use the **ExInterlocked*Xxx*** and the **Interlocked*Xxx*** routines.
- Test every driver on as many different hardware configurations as possible. Always test drivers on multiprocessor systems to find errors that are related to locking, multi-threading, and concurrency.
- Use Driver Verifier (verifier.exe) to test for IRQL and synchronization issues. Use the Forced IRQL Checking option to ensure that spin locks are not used at the wrong IRQL.
- Use the Driver Verifier global counters to monitor IRQL raises and spin lock acquisitions.
- Use Call Usage Verifier (CUV) to check whether the spin locks are allocated and used consistently.

18 Call to Action and Resources

- Take into account multiprocessor issues when you design and test drivers.
- Design drivers to minimize the need for locks.
- Follow the best practices for driver synchronization described in this paper.
- For more information about IRQL issues for drivers, see the companion paper “Scheduling, Thread Context, and IRQL,” at http://www.microsoft.com/whdc/hwdev/driver/IRQL.mspx.
- For other related information, see:
    - “Memory Barriers on Multiprocessor Architectures” at http://www.microsoft.com/whdc/hwdev/driver/mpmem-barrier.mspx.
    - “Interrupt Architecture Enhancements in Microsoft Windows Vista” at http://www.microsoft.com/whdc/hwdev/bus/pci/MSI.mspx.
    - Microsoft Windows Driver Development Kit (DDK) at http://www.microsoft.com/ddk/.
    - Inside Microsoft Windows 2000, Third Edition. Solomon, David A. and Mark Russinovich. Redmond, WA: Microsoft Press, 2000.
    - Designed for Microsoft Windows XP Application Specification at http://www.microsoft.com/winlogo/software/windowsxp-sw.mspx.
    - Microsoft Windows Logo Program System and Device Requirements, Version 2.1a at http://www.microsoft.com/winlogo/hardware/default.mspx.