Portable Systems Group

NT OS/2 Kernel Specification

Author: *David N. Cutler,*
Bryan M. Willman

Original Draft 1.0, March 8, 1989
Revision 1.1, March 16, 1989
Revision 1.2, March 29, 1989
Revision 1.3, April 18, 1989
Revision 1.4, May 4, 1989
Revision 1.5, May 8, 1989
Revision 1.6, August 14, 1989
Revision 1.7, November 15, 1989
Revision 1.8, November 16, 1989
Revision 1.9, November 17, 1989
Revision 1.10, January 6, 1990
Revision 1.11, June 6, 1990
Revision 1.12, September 19, 1990
Revision 1.13, March 11, 1991
Revision 1.14, May 2, 1991
Revision 1.15, May 28, 1991
Revision 1.16, June 18, 1991
Revision 1.17, August 7, 1991
Revision 1.18, August 8, 1991

1. Overview 1
1.1 Kernel Execution Environment 1
1.2 Kernel Use of Hardware Priority Levels 2
1.3 Primary Kernel Data Structures 3
1.4 Multiprocessor Synchronization 4
1.4.1 Executive Multiprocessor Synchronization 6
1.4.1.1 Acquire Executive Spin Lock 6
1.4.1.2 Release Executive Spin Lock 7
1.5 Dispatching 7
1.5.1 Dispatcher Database 8
1.5.2 Idle Thread 8

2. Kernel Objects 9
2.1 Dispatcher Objects 9
2.1.1 Event Object 9
2.1.1.1 Initialize Event 11
2.1.1.2 Pulse Event 11
2.1.1.3 Read State Event 12
2.1.1.4 Reset Event 12
2.1.1.5 Set Event 12
2.1.2 Mutual Exclusion Objects 13
2.1.2.1 Mutant Object 14
2.1.2.1.1 Initialize Mutant 14
2.1.2.1.2 Read State Mutant 15
2.1.2.1.3 Release Mutant 15
2.1.2.2 Mutex Object 16
2.1.2.2.1 Initialize Mutex 17
2.1.2.2.2 Read State Mutex 17
2.1.2.2.3 Release Mutex 18
2.1.2.2.4 Mutex Contention Data 19
2.1.3 Semaphore Object 19
2.1.3.1 Initialize Semaphore 20
2.1.3.2 Read State Semaphore 20
2.1.3.3 Release Semaphore 20
2.1.4 Thread Object 21
2.1.4.1 Initialize Thread 24
2.1.4.2 Alert Thread 26
2.1.4.3 Alert and Resume Thread 27
2.1.4.4 Confine Thread 27
2.1.4.5 Delay Execution 28
2.1.4.6 Disable Queuing of APCs 29
2.1.4.7 Enable Queuing of APCs 29
2.1.4.8 Force Resumption of Thread 30

2.1.4.9 Freeze Thread 30
2.1.4.10 Query Data Alignment Mode 31
2.1.4.11 Query Base Priority 32
2.1.4.12 Read State Thread 32
2.1.4.13 Ready Thread 32
2.1.4.14 Resume Thread 33
2.1.4.15 Rundown Thread 33
2.1.4.16 Set Affinity Thread 34
2.1.4.17 Set Data Alignment Mode 34
2.1.4.18 Set Base Priority 35
2.1.4.19 Set Priority Thread 35
2.1.4.20 Suspend Thread 36
2.1.4.21 Terminate Thread 37
2.1.4.22 Test Alert Thread 37
2.1.4.23 Unfreeze Thread 38
2.1.4.24 Thread Performance Data 39
2.1.5 Timer Object 39
2.1.5.1 Initialize Timer 39
2.1.5.2 Cancel Timer 40
2.1.5.3 Read State Timer 40
2.1.5.4 Set Timer 40
2.2 Control Objects 41
2.2.1 Asynchronous Procedure Call (APC) Object 41
2.2.1.1 Initialize APC 43
2.2.1.2 Flush Queue APC 45
2.2.1.3 Insert Queue APC 46
2.2.1.4 Remove Queue APC 47
2.2.2 Deferred Procedure Call (DPC) Object 47
2.2.2.1 Initialize DPC 48
2.2.2.2 Insert Queue DPC 49
2.2.2.3 Remove Queue DPC 49
2.2.3 Device Queue Object 50
2.2.3.1 Initialize Device Queue 51
2.2.3.2 Insert Device Queue 51
2.2.3.3 Insert By Key Device Queue 52
2.2.3.4 Remove Device Queue 52
2.2.3.5 Remove Entry Device Queue 53
2.2.4 Interrupt Object 54
2.2.4.1 Initialize Interrupt 55
2.2.4.2 Connect Interrupt 58
2.2.4.3 Disconnect Interrupt 58
2.2.4.4 Synchronize Execution 59
2.2.5 Power Notify Object 60

2.2.5.1 Initialize Power Notify 61
2.2.5.2 Insert Power Notify 61
2.2.5.3 Remove Power Notify 62
2.2.6 Power Status Object 62
2.2.6.1 Initialize Power Status 63
2.2.6.2 Insert Power Status 63
2.2.6.3 Remove Power Status 64
2.2.7 Process Object 64
2.2.7.1 Initialize Process 65
2.2.7.2 Attach Process 66
2.2.7.3 Detach Process 67
2.2.7.4 Exclude Process 67
2.2.7.5 Include Process 68
2.2.7.6 Set Priority Process 68
2.2.7.7 Process Accounting Data 69
2.2.8 Profile Object 69
2.2.8.1 Initialize Profile 70
2.2.8.2 Start Profile 70
2.2.8.3 Stop Profile 71
2.2.8.4 Set System Profile Interval 71
2.2.8.5 Query System Profile Interval 72

3. Wait Operations 72
3.1 Wait For Multiple Objects 73
3.2 Wait For Single Object 75

4. Miscellaneous Operations 77
4.1 Bug Check 77
4.2 Context Frame Manipulation 78
4.2.1 Move Machine State To Context Frame 78
4.2.2 Move Machine State From Context Frame 79
4.3 Fill Entry Translation Buffer 80
4.4 Flush Data Cache 81
4.5 Flush Entire Translation Buffer 82
4.6 Flush Instruction Cache 82
4.7 Flush I/O Buffers 83
4.8 Flush Single Translation Buffer Entry 84
4.9 Freeze Execution 86
4.10 Get Current APC Environment 86
4.11 Get Current IRQL 86
4.12 Get Previous Mode 86
4.13 Lower IRQL 87
4.14 Query System Time 87
4.15 Raise IRQL 87

4.16 Run Down Thread 88
4.17 Set System Time........ 88
4.18 Stall Execution........ 89
4.19 Unfreeze Execution 89

5. Intel x86 Specific Functions. 90
5.1 Load an Ldt for a process. 90
5.2 Set and Entry in a Process's Ldt. 90
5.3 Get an Entry from a Thread's Gdt........ 91

1. Overview

This specification describes the kernel layer of the **NT OS/2** operating system. The kernel is responsible for thread dispatching, multiprocessor synchronization, hardware exception handling, and the implementation of low-level machine dependent functions.

The kernel is used by the executive layer of the system to synchronize its activities and to implement the higher levels of abstraction that are exported in user-level API's.

Generally speaking, the kernel does not implement any policy since this is the province of the executive. However, there are some places where policy decisions are made by the kernel. These include the way in which thread priority is manipulated to maximize responsiveness to dispatching events (e.g., the input of a character from a keyboard).

The kernel executes entirely in kernel mode and is nonpageable. It guards access to critical data by raising the processor Interrupt Request Level (IRQL) to an appropriate level and then acquiring a spin lock.

The primary functions provided by the kernel include:

- Support of kernel objects
- Trap handling and exception dispatching
- Interrupt handling and dispatching
- Multiprocessor coordination and context switching
- Power failure recovery
- Miscellaneous hardware-specific functions

It is estimated that the kernel will be less than 48k bytes of resident nonpageable code exclusive of the IEEE exception handling code.

1.1 Kernel Execution Environment

The kernel executes in the most privileged processor mode, usually at an Interrupt Request Level (IRQL) of DISPATCH_LEVEL. The most privileged processor mode is termed kernel mode.

> \ *On the N10 and the x86 architectures the most privileged processor mode is called supervisor mode. However, in other architectures (e.g., MIPS), the*

> *most privileged processor mode is not called supervisor mode. Furthermore, still other architectures include a supervisor mode, but it is not the most privileged mode. Therefore, since it is intended that NT OS/2 be portable and capable of running across several architectures, the most privileged processor mode will be referred to as kernel mode.* \

The kernel can execute simultaneously on all processors in a multiprocessor configuration and synchronize access to critical regions as appropriate.

Software within the kernel is not preemptible and, therefore, cannot be context switched, whereas all software outside the kernel is almost always preemptible and context switchable. In general, executive software outside the kernel is not allowed to raise the IRQL above APC_LEVEL. However, device drivers and executive spin lock synchronization are exceptions to this rule.

The kernel is not pageable and cannot take page faults.

Software within the kernel is written in **C** and assembly language. Assembly language is used for:

- Trap handling
- Spin locks
- Context switching
- Interval timer interrupt
- Power failure interrupt
- Interprocessor interrupt
- I/O Interrupt dispatching
- Machine check processing
- Asynchronous Procedure Call dispatching
- Deferred Procedure Call dispatching
- A small piece of thread startup
- A small piece of system initialization

It is estimated that the number of lines of assembly code within the kernel will be less than 3k.

1.2 Kernel Use of Hardware Priority Levels

Hardware Interrupt Request Levels (IRQL's) are used to prioritize the execution of the various kernel components. IRQL's are hierarchically ordered and each distinct level disables interrupts on lower levels while the respective level is active. The IRQL is raised when hardware and software interrupt requests are granted and by the kernel when synchronization with the possible occurrence of an interrupt is desired.

The kernel uses the hardware Interrupt Request Levels (IRQL's) as follows:

LOW_LEVEL - Thread execution

APC_LEVEL - Asynchronous Procedure Call interrupt

DISPATCH_LEVEL - Dispatch and Deferred Procedure Call interrupt

WAKE_LEVEL - Wake system debugger interrupt

Device levels - Device interrupts

CLOCK2_LEVEL - Interval timer clock interrupt

IPI_LEVEL - Interprocessor interrupt

POWER_LEVEL - Power failure interrupt

HIGH_LEVEL - Machine check and bus error interrupts

The level LOW_LEVEL is reserved for normal thread execution and enables all other interrupts.

The levels APC_LEVEL and DISPATCH_LEVEL are software interrupts and are requested only by the kernel itself. They are located below all hardware interrupt priority levels.

The level WAKE_LEVEL may or may not be present depending on the host hardware configuration and capabilities. It is intended for use in notifying the kernel debugger.

Device interrupt levels are generally placed between the levels WAKE_LEVEL and CLOCK2_LEVEL.

The levels CLOCK2_LEVEL, IPI_LEVEL, POWER_LEVEL, AND HIGH_LEVEL are the highest priority levels and are the most time critical.

\ *The exact specification of interrupt levels is dependent on the host system architecture. The above discussion only defines the importance of the various levels, and does not attempt to assign a numeric value of each level.* \

1.3 Primary Kernel Data Structures

The primary kernel data structures include:

- Interrupt Dispatch Table (IDT) - This is a software maintained table that associates an interrupt source with an Interrupt Service Routine (ISR).

- Processor Control Registers (PCR's) - This is a set of four registers that appear in the same physical address on each processor in a multiprocessor configuration. These registers hold a pointer to the Processor Control Block (PRCB), a pointer to the current thread's Thread Environment Block (TEB), a pointer to the currently active thread, and a temporary location used by the trap handler to save the contents of the stack pointer. On a single processor implementation the PCR is located in main memory.

- Processor Control Block (PRCB) - This structure holds per processor information such as a pointer to the next thread selected for execution on the respective processor. There is a PRCB for each processor in a multiprocessor configuration. The address of this structure can always be obtained from a fixed virtual address on any processor.

- An array of pointers to PRCB's - This array is used to address the PRCB of another processor. It is used when another processor must be interrupted to performed some desired operation.

- Kernel objects - These are the data abstractions that are necessary to control processor execution and synchronization (e.g., thread object, mutex object, etc.). Functions are provided to initialize and manipulate these objects in a synchronized fashion.

- Dispatcher database - This is the database that is required to record the execution state of processors and threads. It is used by the thread dispatcher to schedule the execution of threads on processors.

- Timer queue - This is a list of timers that are due to expire at some future point in time. The timer queue is actually implemented as a splay tree (nearly balanced binary tree maintained by splay transformations).

- Deferred Procedure Call (DPC) queue - This is a list of requests to call a specified procedure when the IRQL falls below DISPATCH_LEVEL.

- **o** Power restoration notify and status queues - These are lists of power notify and status objects that are to be acted upon if power fails and is later restored without the contents of volatile memory being lost.

1.4 Multiprocessor Synchronization

At various stages during its execution, the kernel must guarantee that one, and only one, processor at a time is active within a given critical region. This is necessary to prevent code executing on one processor from simultaneously accessing and modifying data that is being accessed and modified from another processor. The mechanism by which this is achieved is called a *spin lock*.

Spin locks are used when mutual exclusion must exist across all processors and context switching cannot take place. A spin lock takes its name from the fact that, while waiting on the spin lock, software continually tries to gain entry to a critical region and makes no progress until it succeeds.

Spin locks are implemented with a test and set operation on a lock variable. When software executes a test and set operation and finds the previous state of the lock variable free, entry to the associated critical region is granted. If, however, the previous state of the lock variable is busy, then the test and set operation on the lock variable is simply repeated until the previous state is found to be free.

> \ *The exact instructions that are used to implement spin locks are processor architecture specific. In most architectures the test and set operation is not repeated continuously, but rather once finding the lock busy, ordinary instructions are used to poll the lock until it is free. Another test and set operation is then performed to retest the lock. This guarantees a minimum of bus contention during spin lock sequences.* \

Spin locks can only be operated on from a safe interrupt request level. This means that any attempt to acquire a particular spin lock must be at the highest IRQL from which any other attempt to acquire the same spin lock could be made on the same processor. If this restriction were not followed, then deadlock could occur when code running at a lower IRQL acquired a spin lock and then was interrupted by a higher-level interrupt whose Interrupt Service Routine (ISR) also attempted to acquire the spin lock.

The kernel uses various spin locks to synchronize access to the objects and data structures it supports. These include:

- **o** Dispatcher Database - The dispatcher database describes the scheduling state of the system. Whenever a change is made to the dispatching state of the system (e.g., the occurrence of an event), the dispatcher database spin lock must be acquired at IRQL DISPATCH_LEVEL.

- **o** Power Restoration Notify Queue - The power restoration notify queue enables a device driver to be asynchronously notified when power is restored after a failure. Whenever an insertion or removal is made to/from this queue, the power notify queue spin lock must be acquired at IRQL DISPATCH_LEVEL.

- **o** Power Restoration Status Queue - The power restoration status queue provides the capability to have a specified boolean variable set to a value of TRUE when power is restored after a failure. Whenever an insertion or removal is made to/from this queue, the power status queue spin lock must be acquired at IRQL POWER_LEVEL.

- **o** Device Queues - A device queue is used to pass an I/O Request Packet (IRP) between a thread and a device driver. Whenever an insertion or removal is made to/from a device queue, the associated device queue spin lock must be acquired at IRQL DISPATCH_LEVEL.

- **o** Interrupts - Each connected interrupt object has a spin lock that prevents the associated Interrupt Service Routine (ISR) from executing at the same time as other device driver code that accesses the same device resources. Whenever an interrupt occurs and the ISR executes, the associated spin lock must be acquired at the IRQL of the interrupting source. Likewise, device driver code must acquire the associated spin lock at the IRQL of the interrupting source when synchronization with the ISR is required.

- **o** Processor Request Queue - Each processor has a request queue that is used by other processors to signal an action to be performed. Whenever an entry is inserted into or removed from this queue, the associated processor request queue spin lock must be acquired at IRQL IPI_LEVEL.

- **o** Kernel Debugger - The kernel debugger is used to debug the kernel which can be in execution on several processors simultaneously. Whenever the debugger is entered, the kernel debugger spin lock must be acquired at IRQL HIGH_LEVEL.

/ The actual implementation of spin locks may be optimized in a uniprocessor system. This could be done by either generating the system specifically for a uniprocessor system with conditionalized code or by dynamically modifying the code at boot time such that only IRQL is used to synchronize kernel execution. /

1.4.1 Executive Multiprocessor Synchronization

Executive software outside the kernel also has the requirement to synchronize access to resources in a multiprocessor environment. Unlike the kernel, however, executive software can use kernel dispatcher objects (e.g., mutexes, semaphores, events, etc.), as well as spin locks, to implement mutually exclusive access.

Kernel dispatcher objects allow the processor to be redispatched (context switched) and should be used when the wait or access time to a resource is liable to be lengthy (e.g. greater than 25 microseconds on an i860). Spin locks should be used when the wait or access time to a resource is short and does not involve complicated interactions with other code.

Executive spin locks could cause serious maintenance problems if not used judiciously. In particular, no deadlock protection is performed and dispatching is disabled while the executive owns a spin lock. Therefore, certain rules must be followed by executive software when using spin locks:

1. The code within a critical region that is guarded by an executive spin lock must not be pageable and must not make any references to pageable data.

2. An executive spin lock can only be acquired from IRQL's 0, APC_LEVEL, and DISPATCH_LEVEL.

3. The code within a critical region that is guarded by an executive spin lock cannot call any external procedures, nor can it generate any software conditions or hardware exceptions.

Programming interfaces that support executive spin locks include:

KeAcquireSpinLock - Acquire an executive spin lock
KeReleaseSpinLock - Release an executive spin lock.

1.4.1.1 Acquire Executive Spin Lock

An executive spin lock can be acquired with the **KeAcquireSpinLock** function:

```
VOID
KeAcquireSpinLock (
    IN PKSPIN_LOCK SpinLock,
    OUT PKIRQL OldIrql
    );
```

Parameters:

SpinLock - A pointer to an executive spin lock.

OldIrql - A pointer to a variable that receives the previous IRQL.

The previous IRQL is saved in the *OldIrql* parameter, the current IRQL is raised to DISPATCH_LEVEL, and the specified spin lock is acquired. The previous IRQL value must be specified when the spin lock is released.

1.4.1.2 Release Executive Spin Lock

An executive spin lock can be released with the **KeReleaseSpinLock** function:

VOID
KeReleaseSpinLock (
    IN PKSPIN_LOCK *SpinLock*,
    IN KIRQL *OldIrql*
    );

<u>Parameters:</u>

SpinLock - A pointer to an executive spin lock.

OldIrql - The IRQL at which the executive spin lock was acquired.

The specified spin lock is released and the current IRQL is set to the specified value.

1.5 Dispatching

The kernel dispatches threads for execution according to their software priority level.

There are 32 levels of thread priority which are split into two classes:

- Realtime
- Variable

The priority of threads within the realtime priority class is not altered by the kernel. However, as quantum end events occur, the threads within a level are round robin scheduled.

The priority of threads within the variable priority class is altered by the kernel, dependent on the execution profile of the respective threads. At each quantum end event, the priority of the executing thread is decremented and a decision is made as to whether it should be preempted by another thread. If it should be preempted to execute a higher priority thread, then a context switch occurs. When a thread in the variable priority class transitions from a Waiting state to a Ready state, it is given a priority boost that is dependent on the type of event that caused the Wait to be satisifed. If the event was keyboard input, for instance, the thread would get a large boost. However, if the event was file I/O it would be given a smaller boost.

When a thread is readied for execution, an attempt is made to dispatch the thread on an idle processor. If an idle processor can be selected, then preference is given to the last processor on which the thread executed.

If an idle processor is not available, then an attempt is made to find a processor that should be preempted. This determination is made using the active summary and the active matrix (these structures are described in the following two sections). If an appropriate processor is located, then preference is given to the last processor on which the thread executed.

If no processor can be preempted to execute the ready thread, the thread is inserted at the end of the ready queue selected by its priority and the ready summary is updated.

Giving preference to the last processor a thread executed on maximizes the chances there is still thread data in the respective processor's secondary cache.

1.5.1 Dispatcher Database

The kernel maintains several data structures to aid in choosing which threads should be active at any instance in time. These structures include:

- Ready queues - There is a ready queue for each software priority level. Each queue contains a list of threads that are ready to execute at that level.
- Ready summary - A set that contains a TRUE member for each ready queue that contains one or more threads.
- Active matrix - The active matrix is a two-dimensional array that represents a set of processors for each of the software priority levels. A member is TRUE in the matrix if a processor is executing a thread at the corresponding priority level.
- Active summary - A set that contains a TRUE member for each priority class that has one or more processors executing threads at that level.
- Idle summary - A set that contains a TRUE member for each processor that is currently idle.
- Idle thread - A thread that is run when no other thread is available to execute on a processor.

The ready summary is used to quickly locate a thread to execute when the currently executing thread is terminated or transitions to a Waiting state.

The active summary is used to quickly determine if preemption should occur when a thread transitions to a Ready state.

> */ Since this determination is simple in a uniprocessor system, the active summary and active matrix are only kept up to date and used on configurations with multiple processors. /*

1.5.2 Idle Thread

Each processor has an idle thread that can always execute. The idle thread has a stack that is capable of nesting all interrupts and a software priority that is below that of all other thread priority levels.

The idle thread is selected for execution when no other thread is available to execute. The idle thread executes at DISPATCH_LEVEL and continually loops looking for work that has been assigned to its processor. This work includes processing the Deferred Procedure Call (DPC) queue and initiating a context switch when another thread is selected for execution on the respective processor.

While an idle thread executes, the member in the idle summary selected by its processor number is TRUE. This enables the kernel to quickly determine which processors are executing idle threads.

2. Kernel Objects

The kernel exports a set of abstractions to the executive layer which are called *kernel objects*. Kernel objects are represented by a control block that describes the contents of each object. Kernel objects are used by the executive layer to construct more complex objects that are exported in user level API's.

There are two types of kernel objects:

1. *Dispatcher* objects

2. *Control* objects

Dispatcher objects have a signal state (*Signaled* or *Not-Signaled*) and control the dispatching and synchronization of system operations. These objects include the *event*, *mutant, mutex*, *semaphore*, *thread*, and *timer* objects. Dispatcher objects can be specified as arguments to the kernel Wait functions.

Control objects are used to control the operation of the kernel but do not affect dispatching or synchronization. These objects include the *Asynchronous Procedure Call* (APC), *Deferred Procedure Call* (DPC), *device queue*, *interrupt*, *power notify*, *power status*, *process*, and *profile* objects.

The kernel neither allocates nor deallocates kernel object storage. It is the responsibility of the executive layer to allocate an appropriate data structure and call the kernel to initialize a specific kernel object type.

The kernel exports kernel object types to the executive layer so the executive can allocate appropriately sized data structures and can access various read only data items (e.g., linkage pointers).

The executive is not allowed to manipulate the writeable data portion of kernel objects directly. Various interfaces are provided by the kernel to perform this type of operation.

Kernel objects are referenced by pointers to the respective data structures. It is the responsibility of the executive layer to synchronize the deallocation of kernel object storage such that the kernel does not access an object after its storage has been deleted.

2.1 Dispatcher Objects

This section describes the various types of dispatcher objects and the interfaces that are provided to manipulate these objects.

2.1.1 Event Object

An *event object* is used to record the occurrence of an event and synchronize it with some action that is to be performed.

There are two types of event objects:

- *synchronization*
- *notification*

A synchronization event object is used when it is desirable for a single waiter to continue execution when the event is set to the Signaled state. The state of a synchronization event object is automatically reset to the Not-Signaled state when a Wait for the event object is satisfied.

Synchronization events provide an optimal way to implement mutual exclusion at user level. An identical capability can be implemented using binary semaphores, but requires calling the system each time mutual exclusion is desired.

Synchronization events can also be used to provide synchronization in producer/consumer relationships where it is otherwise undesirable to use a counting semaphore.

A notification event is used when it is desirable for all waiters to continue execution when the event is set to the Signaled state. The state of a notification event object is not altered when a Wait for the event object is satisfied and remains Signaled until it is explicitly reset to the Not-Signaled state.

Notification events can be used to implement resource allocators where there is not a one-to-one relationship between the release of a resource and the allocation of the resource (e.g. a memory allocator).

Waiting on an event object causes the execution of the subject thread to be suspended until the event object attains a state of Signaled.

Satisfying the Wait for a synchronization event object automatically causes the state of the event object to be reset to the Not-Signaled state.

Satisfying the Wait for a notification event object does not cause the state of the event object to change. Therefore, when a notification event object attains a state of Signaled, an attempt is made to satisfy as many Waits as possible.

The state of an event object is controlled by a count value that is incremented each time the event object is set to the Signaled state. Thus, the state of the event object is Signaled when the count value is nonzero and Not-Signaled when the count value is zero.

The count value indicates the number of times that the event object has been set to a Signaled state since the last time it was reset to the Not-Signaled state.

Programming interfaces that support the event object include:

KeInitializeEvent - Initialize an event object
KePulseEvent - Set/reset event object state atomically
KeReadStateEvent - Read state of event object
KeResetEvent - Set event object to Not-Signaled state
KeSetEvent - Set event object to Signaled state

2.1.1.1 Initialize Event

An event object can be initialized with the **KeInitializeEvent** function:

VOID
KeInitializeEvent (
 IN PKEVENT *Event*,
 IN EVENT_TYPE *EventType*,
 IN BOOLEAN *State*
 );

<u>Parameters:</u>

Event - A pointer to a dispatcher object of type event.

EventType - The event type (*NotificationEvent* or *SynchronizationEvent*).

State - The initial state of the event.

The event object data structure for the specified event type is initialized with the specified initial state.

2.1.1.2 Pulse Event

An event object can be atomically set to a Signaled state and then reset to a Not-Signaled state with the **KePulseEvent** function:

LONG
KePulseEvent (
 IN PKEVENT *Event*,
 IN KPRIORITY *Increment*,
 IN BOOLEAN *Wait*
 );

<u>Parameters:</u>

Event - A pointer to a dispatcher object of type event.

Increment - The priority increment that is to be applied if setting the event causes a Wait to be satisfied.

Wait - A boolean value that specifies whether the call to **KePulseEvent** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

The function atomically sets the state of the event object to Signaled, attempts to satisfy as many Waits as possible for the event object, and then resets the state of the event object to Not-Signaled.

The previous state of the event object is returned as the function value. If the previous state of the event object was Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

If the *wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KePulseEvent MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to set an event and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.1.3 Read State Event

The current state of an event object can be read with the **KeReadStateEvent** function:

LONG
KeReadStateEvent (
    IN PKEVENT *Event*
    );

<u>Parameters:</u>

Event - A pointer to a dispatcher object of type event.

The current state of the event object is returned as the function value. If the current state of the event object is Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

2.1.1.4 Reset Event

An event object can be reset to a Not-Signaled state with the **KeResetEvent** function:

LONG
KeResetEvent (
    IN PKEVENT *Event*
    );

<u>Parameters:</u>

Event - A pointer to a dispatcher object of type event.

The previous state of the event object is returned as the function value and the state of the event object is reset to Not-Signaled (i.e., the count value is set to zero). If the previous state of the event object was Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

2.1.1.5 Set Event

An event object can be set to a Signaled state with the **KeSetEvent** function:

LONG
KeSetEvent (
    IN PKEVENT *Event*,
    IN KPRIORITY *Increment*,
    IN BOOLEAN *Wait*
    );

Parameters:

Event - A pointer to a dispatcher object of type event.

Increment - The priority increment that is to be applied if setting the event causes a Wait to be satisfied.

Wait - A boolean value that specifies whether the call to **KeSetEvent** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

The previous state of the event object is returned as the function value and the state of the event is set to Signaled (i.e., the count value is incremented). If the previous state of the event object was Signaled, then a nonzero count value is returned. Otherwise, a value of zero is returned.

Setting an event object causes the event to attain a Signaled state, and therefore, an attempt is made to satisfy as many Waits as possible for the event object.

If the *Wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spinlock. Thus the call to **KeSetEvent MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to set an event and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.2 Mutual Exclusion Objects

The kernel provides two objects for controlling mutually exclusive access to a resource; the *mutant object* and the *mutex object*.

The *mutant object* is intended for use in providing a user mode mutual exclusion mechanism that has ownership semantics, but it can also be used in kernel mode.

The *mutex object* can only be used in kernel mode and is intended to provide a deadlock-free mutual exclusion mechanism with ownership and other special system semantics.

The state of mutant and mutex objects is controlled by a count. When the count is one, the mutant or mutex object is in the Signaled state, the mutant or mutex object is not owned, and exclusive access to the corresponding resource can be obtained by specifying the mutant or mutex object in a kernel Wait function. When the count is not one, the mutant or mutex object is in the Not-Signaled state and any attempt to acquire the mutant or mutex object will cause the subject thread to wait until the mutant or mutex object count is one.

Mutant and mutex objects are similar in that they both provide mutual exclusion mechanisms with recursive ownership capability. They have significant differences, however, which dictate the support of two separate object types. These differences include the following:

- Mutex objects have a level number which is used to prevent deadlock, whereas mutant objects have no level number.
- Mutant objects have an abandoned status and can be released by a thread other than the owner, whereas mutex objects do not have an abandoned status and can only be released by the owner thread.
- Owning a mutex object prevents the owning thread's process from leaving the balance set, whereas owning a mutant object does not affect the swapability of the parent process.
- Owning a mutex object causes the priority of the owning thread to be raised to the greater of its current priority and the lowest realtime priority, whereas owning a mutant object does not affect the owner thread's priority in any way.
- Owning a mutex object prevents the delivery of kernel mode APCs, whereas owning a mutant object does not affect the delivery of kernel mode APCs.

2.1.2.1 Mutant Object

Waiting on (acquiring) a mutant object causes the execution of the subject thread to be suspended until the mutant object attains a state of Signaled. Satisfying the Wait for a mutant object causes the state of the mutant object to become Not-Signaled.

Threads are allowed to recursively acquire mutant objects that they already own. A recursively acquired mutant object must be released the same number of times it was acquired before it will again attain the state of Signaled.

Mutant objects can be exported to user mode for providing a mutual exclusion mechanism with ownership semantics.

Programming interfaces that support the mutant object include:

KeInitializeMutant - Initialize a mutant object
KeReadStateMutant - Read the state of a mutant object
KeReleaseMutant - Release ownership of a mutant object

2.1.2.1.1 Initialize Mutant

A mutant object can be initialized with the **KeInitializeMutant** function:

```
VOID
KeInitializeMutant (
    IN PKMUTANT Mutant,
    IN BOOLEAN InitialOwner
    );
```

Parameters:

Mutant - A pointer to a dispatcher object of type mutant.

InitialOwner - A boolean variable that determines whether the current thread is to be the initial owner of the mutant object.

If the value of the *InitialOwner* parameter is TRUE, then the mutant object data structure is initialized with the current thread as the owner and an initial state of Not-Signaled. Otherwise, the mutant object data structure is initialized as unowned with an initial state of Signaled.

2.1.2.1.2 Read State Mutant

The current state of a mutant object can be read with the **KeReadStateMutant** function:

LONG
KeReadStateMutant (
    IN PKMUTANT *Mutant*
    );

Parameters:

Mutant - A pointer to a dispatcher object of type mutant.

The current state of the mutant object is returned as the function value. If the return value is one, then the state of the mutant object is Signaled. Otherwise, the state of the mutant object is Not-Signaled.

2.1.2.1.3 Release Mutant

A mutant object can be released with the **KeReleaseMutant** function:

LONG
KeReleaseMutant (
    IN PKMUTANT *Mutant*,
    IN KPRIORITY *Increment*,
    IN BOOLEAN *Abandoned*,
    IN BOOLEAN *Wait*
    );

Parameters:

Mutant - A pointer to a dispatcher object of type mutant.

Increment - The priority increment that is to be applied if releasing the mutant object causes a Wait to be satisfied.

Abandoned - A boolean value that specifies whether the release of the mutant object is to be forced.

Wait - A boolean variable that specifies whether the call to **KeReleaseMutant** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

If the value of the *Abandoned* parameter is TRUE, then the release of the mutant object is unconditional and can be requested by a thread other than the owner of the mutant object. The fact that the mutant object is being abandoned is recorded in the mutant object and is returned by the kernel Wait services when ownership of the mutant object is granted to another thread. Once set, the abandoned status of a mutant object cannot be cleared and is thereafter always returned by the kernel Wait services.

If the value of the *Abandoned* parameter is FALSE, then only the owning thread can release the mutant object. Any attempt to release the mutant object made by a thread other than the owner, will cause an exception to be raised. If the mutant object has previously been abandoned, then the exception STATUS_ABANDONED is raised. Otherwise, the exception STATUS_MUTANT_NOT_OWNED is raised.

The previous state of the mutant object is returned as the function value.

If the value of the *Abandoned* parameter is TRUE, then the state of the mutant object is set to Signaled and the return value is not meaningful.

If the value of the *Abandoned* parameter is FALSE, then the new state of the mutant object can be determined by the value returned. If the returned value is zero, then the mutant object was actually released and attained a state of Signaled. Otherwise, the mutant object was not released and still has a state of Not-Signaled (i.e., the mutant object has been recursively acquired and has not yet been released the proper number of times to cause it to attain a Signaled state).

If the mutant object attains a Signaled state, then an attempt is made to satisfy a Wait for the mutant object.

If the mutant object attains a Signaled state and was previously owned by a thread, then the mutant object is removed from the list of mutant objects owned by the subject thread.

If the value of the *Wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KeReleaseMutant MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to release a mutant object and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.2.2 Mutex Object

Waiting on (acquiring) a mutex object causes the execution of the subject thread to be suspended until the mutex object attains a state of Signaled. Satisfying the Wait for a mutex object causes the state of the mutex object to become Not-Signaled and disables the delivery of normal kernel APCs to the subject thread.

If the subject thread did not previously own any mutexes, then the current execution priority of the thread is saved and then raised to the maximum of its current priority and the lowest realtime priority. This ensures that the thread will have a very high execution priority while it executes in a critical section and prevents the effects of priority inversion. When the thread releases the last mutex it owns, its priority is restored to the saved value.

Mutex ownership also prevents the owning thread's process from being removed from the balance set. If the balance set manager selects the process for removal from the balance set, all threads within the process that own mutexes will be allowed to continue execution until they no longer own any mutexes. This ensures that access to critical resources is not blocked because a thread belonging to a process that has been removed from the balance set owns one or more mutexes.

Each mutex object has a level number. This level number is used to prevent possible deadlock. When an attempt is made to acquire a mutex object, the level number of the mutex object must be higher (numerically) than the highest level number of any mutex object owned by the subject thread. If this condition is not met, then a system bug check occurs.

Level number checking is included mainly for debugging the system while it is under development. It may or may not be conditionalized out in a production system.

Threads are allowed to recursively acquire mutex objects that they already own. For this case level number checking does not occur since deadlock is not possible. A recursively acquired mutex object must be released the same number of times it was acquired before it will again attain the state of Signaled.

Mutex objects are not exported by the executive to user mode and are only available for use by the executive layer itself. Furthermore, the executive is not allowed to acquire a mutex object and then return to user mode while retaining ownership of the mutex.

Programming interfaces that support the mutex object include:

KeInitializeMutex - Initialize a mutex object
KeReadStateMutex - Read state of mutex object
KeReleaseMutex - Release ownership of mutex object

2.1.2.2.1 Initialize Mutex

A mutex object can be initialized with the **KeInitializeMutex** function:

VOID
KeInitializeMutex (
    IN PKMUTEX *Mutex*,
    IN ULONG *Level*
    );

Parameters:

Mutex - A pointer to a dispatcher object of type mutex.

Level - The level number that is to be assigned to the mutex.

The mutex object data structure is initialized with the specified level number and an initial state of Signaled.

2.1.2.2.2 Read State Mutex

The current state of a mutex object can be read with the **KeReadStateMutex** function:

LONG
KeReadStateMutex (
    IN PKMUTEX *Mutex*
    );

Parameters:

Mutex - A pointer to a dispatcher object of type mutex.

The current state of the mutex object is returned as the function value. If the return value is one, then the state of the mutex object is Signaled. Otherwise, the state of the mutex object is Not-Signaled.

2.1.2.2.3 Release Mutex

A mutex object can be released with the **KeReleaseMutex** function:

LONG
KeReleaseMutex (
  IN PKMUTEX *Mutex*,
  IN BOOLEAN *Wait*
  );

<u>Parameters:</u>

Mutex - A pointer to a dispatcher object of type mutex.

Wait - A boolean variable that specifies whether the call to **KeReleaseMutex** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

The previous state of the mutex object is returned as the function value. The state is returned as an integer value. If the returned value is zero, then the mutex object was actually released and attained a state of Signaled. Otherwise, the mutex object was not released and still has a state of Not-Signaled (i.e the mutex object has been recursively acquired and has not yet been released the proper number of times to cause it to attain a Signaled state).

If the mutex object attains a Signaled state, then an attempt is made to satisfy a Wait for the mutex object.

A mutex object can only be released by the subject thread that owns the mutex. If an attempt is made to release a mutex that is not owned by the subject thread, then a bug check will occur.

A mutex object can only be released if it is currently owned. An attempt to release a mutex object whose current state is Signaled, will also cause a bug check to occur.

If the mutex object attains a Signaled state, then the mutex object is removed from the list of mutexes owned by the subject thread. If the thread's owned mutex list does not contain any more entries, then the thread's original priority is restored (the priority that was previously saved) and a kernel APC is requested if the thread's kernel APC queue contains one or more entries.

If the mutex object attains a Signaled state, the mutex was the last one owned by the subject thread, and the thread's process has been selected for removal from the balance set, then a new thread is selected for execution, the subject thread is inserted into its process's ready queue, and a context switch to the selected thread is performed. If no other threads in the process own mutexes, then the balance set event is set in the process object to notify the balance set manager that it can remove the process from the balance set.

If the value of the *Wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KeReleaseMutex MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to release a mutex and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.2.2.4 Mutex Contention Data

Two counters are maintained for each mutex object to determine the activity level of the mutex object and any contention that may occur. One of the counters records the number of times the mutex object has been acquired and the other the number of times an attempt to acquire the mutex object resulted in the execution of the subject thread being suspended.

2.1.3 Semaphore Object

A *semaphore object* is used to control access to a resource, but not necessarily in a mutually exclusive fashion. A semaphore object acts as a gate through which a variable number of threads may pass concurrently, up to a specified limit. The gate is open (Signaled state) as long as there are resources available. When the number of resources specified by the limit are concurrently in use, the gate is closed (Not-Signaled state).

The gating mechanism of a semaphore object is controlled by a count. When the count is greater than zero, the semaphore object is in the Signaled state, and one or more threads may pass through the gate by specifying the semaphore in a kernel Wait function. When the count is zero, the semaphore object is in the Not-Signaled state, the gate is closed, and any attempt to pass through the gate will cause the subject thread to Wait until the semaphore count is greater than zero.

Waiting on (acquiring) a semaphore object causes the execution of the subject thread to be suspended until the semaphore object attains a Signaled state. Satisfying the Wait for a semaphore object causes the semaphore count to be decremented.

A semaphore object with a limit of one can be used to provide mutual exclusion semantics since only one thread will be allowed to pass through the gate concurrently. This is not, however, the same functionality as provided by mutex objects since there is no ownership (i.e., any thread can release the semaphore), there is no level number checking (i.e., deadlock is not prevented), and the priority of the subject thread is not raised (i.e., priority inversion problems can arise).

A semaphore object with a limit of one can also be used as a "*synchronization*" event provided the semaphore is never "over Signaled" (i.e., no thread attempts to release the semaphore while it is already in the Signaled state). For this case, the semaphore is normally in a Not-Signaled state and is used to record the occurrence of an event by releasing the semaphore. Waiting on the semaphore object suspends the subject thread until the semaphore attains a Signaled state and causes the semaphore to be immediately set to the Not-Signaled state.

Programming interfaces that support the semaphore object include:

KeInitializeSemaphore - Initialize a semaphore object
KeReadStateSemaphore - Read state of semaphore
KeReleaseSemaphore - Adjust semaphore object count

2.1.3.1 Initialize Semaphore

A semaphore object can be initialized with the **KeInitializeSemaphore** function:

VOID
KeInitializeSemaphore (
  IN PKSEMAPHORE *Semaphore*,
  IN LONG *Count*,
  IN LONG *Limit*
  );

Parameters:

Semaphore - A pointer to a dispatcher object of type semaphore.

Count - The initial count value to be assigned to the semaphore. This value must be positive.

Limit - The maximum count value that the semaphore can attain. This value must be positive.

The semaphore object data structure is initialized with the specified initial count and limit.

2.1.3.2 Read State Semaphore

The current state of a semaphore object can be read with the **KeReadStateSemaphore** function:

LONG
KeReadStateSemaphore (
    IN PKSEMAPHORE *Semaphore*
    );

Parameters:

Semaphore - A pointer to a dispatcher object of type semaphore.

The current signal state of the semaphore object is returned as the function value. If the return value is zero, then the current state of the semaphore object is Not-Signaled. Otherwise, the current state of the semaphore object is Signaled.

2.1.3.3 Release Semaphore

A semaphore object can be released with the **KeReleaseSemaphore** function:

LONG
KeReleaseSemaphore (
    IN PKSEMAPHORE *Semaphore*,
    IN KPRIORITY *Increment*,
    IN LONG *Adjustment*,
    IN BOOLEAN *Wait*
    );

Parameters:

Semaphore - A pointer to a dispatcher object of type semaphore.

Increment - The priority increment that is to be applied if releasing the semaphore causes a Wait to be satisfied.

Adjustment - The value that is to be added to the current semaphore count. This value must be positive.

Wait - A boolean value that specifies whether the call to **KeReleaseSemaphore** will be IMMEDIATELY followed by a call to one of the kernel Wait functions.

Releasing a semaphore object causes the semaphore count to be augmented by the value of the *Adjustment* parameter. If the resultant value is greater than the limit of the semaphore object, then the count is not adjusted and the exception STATUS_SEMAPHORE_COUNT_EXCEEDED is raised.

Augmenting the semaphore object count causes the semaphore to attain a state of Signaled, and therefore, an attempt is made to satisfy as many Waits as possible for the semaphore object.

The previous state of the semaphore object is returned as an integer function value. If the return value is zero, then the previous state of the semaphore object was Not-Signaled. Otherwise, the previous state of the semaphore object was Signaled.

If the value of the *wait* parameter is TRUE, then the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Thus the call to **KeReleaseSemaphore MUST** be IMMEDIATELY followed by a call to one of the kernel Wait functions. This capability is provided to allow the executive to release a semaphore and Wait as one atomic operation which prevents a possible superfluous context switch.

2.1.4 Thread Object

A *thread object* is the agent that executes program code and is dispatched for execution by the kernel.

Each thread is associated with a *process object* which specifies the virtual address space mapping for the thread and accumulates thread execution time. Several thread objects can be associated with a single process object which enables the concurrent execution of multiple threads in a single address space (possibly simultaneous execution in a multiprocessor system).

A thread executes in kernel and user mode, usually at IRQL 0, and is dispatched for execution according to its software priority.

Although there is no actual difference, threads are usually referred to as either user threads or system threads. A user thread executes mostly in user mode and within the user part of the virtual address space. It enters kernel mode only to execute system services. System threads execute only in kernel mode and usually within the system part of the virtual address space. There are some system threads, however, that also use the user part of the address space to store information and execute code from. An example of such a thread is a file system.

The context of a thread typically consists of the following:

- Integer registers
- Floating point registers
- Architecture-dependent special registers

- A user stack pointer
- A kernel stack pointer
- A program counter
- A processor status
- A floating point status

\ *The exact context of a thread is host architecture dependent.* \

Each thread has a set of processors on which it can execute. This is referred to as the processor affinity. When a thread is initialized it is given the processor affinity of its parent process. Thereafter, the affinity of the thread can be set to any proper subset of the parent process's affinity.

A thread can be in one of six dispatcher, or scheduling, states:

- *Initialized*
- *Ready*
- *Standby*
- *Running*
- *Waiting*
- *Terminated*

A thread enters the *Initialized* state when its thread object is initialized. A thread in the Initialized state can transition only to the Ready state.

A thread is in a *Ready* state when it is eligible to be selected for execution on a processor. A thread in the Ready state is enqueued on the dispatcher ready queue selected by its priority and can transition from the Ready state to the Standby state.

A thread is in a *Standby* state when it has been selected to execute on a processor, but the actual context switch to the thread has not yet occurred. A thread in the Standby state can transition to the Ready and Running states.

A thread is in a *Running* state when it is currently being executed by a processor. A thread in the Running state can transition to the Ready, Waiting, and Terminated states.

A thread is in a *Waiting* state when it is waiting for one or more dispatcher objects to attain a state of Signaled. A thread in the Waiting state can transition to the Ready state.

A thread in the *Terminated* state has completed its execution and the corresponding thread object will be deleted by the executive at the appropriate time.

> */ Note that is possible to reuse a thread object that has a state of Terminated by simply reinitializing the thread object which will cause it to enter the Initialized state. /*

A thread's parent process is either in the balance set (Included) or not in the balance set (Excluded). The balance set is that set of processes and threads that are currently eligible for being considered for execution. Processes and threads that are not in the balance set are not considered for execution until they reenter the balance set.

The balance set is managed by the balance set manager. In a single user system there will be no balance set manager. Server systems, however, present the problem of having to manage more processes than there is space for in main memory without incurring excessive paging. Therefore, the balance set manager is responsible for determining when excessive paging is occurring and then selecting the appropriate processes to remove from the balance set.

> */ There may not be a balance set manager in the first release of NT OS/2. We may rely instead on working set trimming to obtain necessary memory when excessive paging levels are observed. /*

A thread is dispatched for execution according to its software priority level. Higher priority threads are given preference and preempt the execution of lower priority threads.

There are two classes of priority: 1) realtime, and 2) variable. Each of these classes contains several levels of thread priority.

In the realtime priority class, a thread executes at a priority level selected by the user. The system makes no attempt to alter or boost the priority as the thread executes and enters and leaves wait states. The realtime priority levels are higher than all the levels in the variable priority class. The realtime priority class is intended for use by time-critical threads that require a response time that is guaranteed by application design.

The variable priority class is where most threads execute. As a thread executes and experiences quantum end events, its priority decays. When a thread enters a Waiting state and is subsequently awakened, it is given a priority boost that is commensurate with the importance of the event that caused the Wait to be satisifed (e.g., a large boost is given for the completion of keyboard input but a small one is given for completing disk I/O). A thread therefore, runs at a high priority as long as it is interactive. When it becomes compute bound, its priority rapidly decays, and it is considered only after other, higher priority threads. In addition, the kernel arbitrarily boosts the priority of threads that are compute bound and haven't received any processor time for a given period of time.

As a thread executes, performance and accounting data are collected for the thread and for the thread's parent process.

Programming interfaces that support the thread object include:

KeInitializeThread - Initialize a thread object
KeAlertThread - Set thread alert for specified mode
KeAlertResumeThread - Alert and resume thread object
KeConfineThread - Confine thread object execution
KeDelayExecutionThread - Delay execution of thread object
KeDisableApcQueuingThread - Disable queuing of APCs
KeEnableApcQueuingThread - Enable queuing of APCs
KeForceResumeThread - Force resumption of thread execution
KeFreezeThread - Freeze thread object execution
KeQueryAutoAlignmentThread - Query alignment mode of thread object
KeQueryBasePriorityThread - Query base priority of thread object
KeReadStateThread - Read state of thread object
KeReadyThread - Ready thread object for execution
KeResumeThread - Resume thread object execution
KeRundownThread - Run down thread object
KeSetAffinityThread - Set thread object processor set
KeSetAutoAlignmentThread - Set alignment mode of thread object
KeSetBasePriorityThread - Set base priority of thread object
KeSetPriorityThread - Set priority of thread object
keSuspendThread - Suspend thread object exection
KeTerminateThread - Terminate thread object execution
KeTestAlertThread - Test if thread alerted for mode
KeUnfreezeThread - Unfreeze thread object execution

2.1.4.1 Initialize Thread

A thread object can be initialized with the **KeInitializeThread** function:

VOID
KeInitializeThread (
    IN PKTHREAD *Thread*,
    IN PVOID *KernelStack*,
    IN PKSYSTEM_ROUTINE *SystemRoutine*,
    IN PKSTART_ROUTINE *StartRoutine* **OPTIONAL**,
    IN PVOID *StartContext* **OPTIONAL**,
    IN PCONTEXT *ContextFrame* **OPTIONAL**,
    IN PVOID *Teb* **OPTIONAL**,
    IN PKPROCESS *Process*
    );

<u>Parameters:</u>

Thread - A pointer to a dispatcher object of type thread.

KernelStack - A pointer to the base (highest address) of a kernel stack on which the initial context for the thread is to be constructed.

SystemRoutine - A pointer to a function that is to called when the thread is scheduled for execution. This routine performs executive initialization.

StartRoutine - An optional pointer to a function that is to be called after the executive has finished initializing the thread.

StartContext - A optional pointer to an arbitrary data structure which will be passed to the *StartRoutine* function as a parameter.

ContextFrame - An optional pointer to a context frame which contains the initial user mode state of the thread. This parameter is specified if the thread will execute in user mode. If this parameter is not specified, then the *Teb* parameter is ignored.

Teb - An optional pointer to the user mode thread environment block. This parameter is ignored if the *ContextFrame* parameter is not specified.

Process - A pointer to a control object of type process.

The function specified by the *SystemRoutine* parameter has the following type definition:

typedef
VOID
(***PKSYSTEM_ROUTINE**) (

    IN PKSTART_ROUTINE *StartRoutine* **OPTIONAL**,
    IN PVOID *StartContext* **OPTIONAL**
    );

Parameters:

StartRoutine - An optional pointer to a function that is to be called after the executive has finished initializing the thread.

StartContext - A optional pointer to an arbitrary data structure which will be passed to the *StartRoutine* function as a parameter.

The function specified by the *StartRoutine* parameter has the following type definition:

typedef
VOID
(***PKSTART_ROUTINE**) (
    IN PVOID *StartContext*
    );

Parameters:

StartContext - A pointer to an arbitrary data structure that was specified when the thread object was initialized.

The thread object data structure is initialized and the thread's dispatcher state is set to Initialized. The thread's quantum, affinity, data alignment handling mode, current priority, and base priority are taken from the parent process object.

A kernel context frame is built on the specified kernel stack which will cause the thread to begin execution in the kernel thread startup routine. The kernel thread startup routine will call the specified start routine which is responsible for initializing the executive state of the thread as necessary. If the thread is a system thread, then the executive startup routine will call the thread's entry point directly. If, however, the thread is a user thread, then the executive startup routine returns control to the kernel thread startup routine which restores the user mode state and continues execution of the thread in user mode.

A thread begins execution in kernel mode at IRQL APC_LEVEL with the queuing of APCs enabled. It is the responsibility of the executive to lower IRQL to 0 as soon as thread initialization is complete.

Once a thread object has been initialized, it can be readied for execution with the **KeReadyThread** function.

2.1.4.2 Alert Thread

A thread object can be alerted with the **KeAlertThread** function:

BOOLEAN
KeAlertThread (
    IN PKTHREAD *Thread*,
    IN KPROCESSOR_MODE *AlertMode*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

AlertMode - The processor mode (*UserMode* or *KernelMode*) for which the thread is to be alerted.

Alerting a thread object causes the alert variable associated with the specified processor mode to be set to a value of TRUE.

If the thread object is currently in a Wait state, the Wait is alertable, and the specified processor mode is less than or equal to the Wait mode, then the thread is Unwaited with a completion status of *STATUS_ALERTED* and the specified alert variable is set to a value of FALSE.

Alerts provide a way in which to break into a thread's execution at well-defined points. These points occur when the thread Waits in an alertable state and when the thread polls the alerted flag using the **KeTestAlertThread** function.

The previous value of the alert variable for the specified processor mode is returned as the function value. If the return value is TRUE, then the subject thread was already alerted for the specified processor mode. If the return value is FALSE, then the subject thread was not previously alerted.

2.1.4.3 Alert and Resume Thread

A thread object can be kernel mode alerted and its execution resumed with the **KeAlertResumeThread** function:

ULONG
KeAlertResumeThread (
    IN PKTHREAD *Thread*
    );

<u>Parameters:</u>

Thread - A pointer to a dispatcher object of type thread.

This function executes the equivalent of a **KeAlertThread** function for kernel mode followed by a **KeResumeThread** function on the specified thread object.

Resuming a thread object checks the suspend count of the subject thread. If the suspend count is zero, then the thread is not currently suspended and no operation is performed. Otherwise, the subject thread's suspend count is decremented. If the resultant value is zero, then the execution of the subject thread is resumed by releasing its builtin suspend semaphore.

The previous suspend count is returned as the function value. If the return value is zero, then the subject thread was not previously suspended. If the return value is one, then the subject thread's execution was resumed. If the returned value is not zero or one, then the subject thread is still suspended and must be resumed the number of times specified by the return value minus one before it will actually resume execution.

2.1.4.4 Confine Thread

The execution of the current thread can be confined to the current processor with the **KeConfineThread** function:

KAFFINITY
KeConfineThread (
    );

Confining the execution of the current thread to the current processor causes the thread's affinity to be set such that it can only execute on the current processor. The previous affinity is returned as the function value and can be used to later restore the thread's affinity with the **KeSetAffinityThread** function.

This function is useful when it is desirable to avoid translation buffer flushes across the entire multiprocessor complex while certain page manipulations are taking place. For example, the zero page writer selects a page to zero, confines its execution to the current processor, flushes the current processor's translation buffer, maps the page into the system part of the virtual address space reserved for zeroing pages, and then proceeds to zero the page. If the execution of the zero page writer was not confined during the page zeroing operation, then the translation buffers of all processors in the multiprocessor complex would have to be flushed before mapping and zeroing the page could commence.

2.1.4.5 Delay Execution

The execution of the current thread can be delayed for a specified interval of time with the **KeDelayExecutionThread** function:

NTSTATUS
KeDelayExecutionThread (
  IN KPROCESSOR_MODE *WaitMode*,
  IN BOOLEAN *Alertable*,
  IN PTIME *Interval*
  );

Parameters:

WaitMode - The processor mode on whose behalf the Wait is occurring.

Alertable - A boolean value that specifies whether the Wait is alertable.

Interval - The absolute or relative time over which the Wait is to occur.

The expiration time is computed and the current thread is put in a Waiting state. When the specified interval of time has passed, the thread will exit the Waiting state and continue execution.

The reason for the Wait is set to DelayExecution.

The *WaitMode* parameter specifies on whose behalf the Wait is occurring (i.e., kernel or user).

The *Alertable* parameter specifies whether the thread can be alerted while it is in the Waiting state. If the value of this parameter is TRUE and the thread is alerted for a mode that is equal to or more privileged than the Wait mode, then the thread's Wait will be satisfied with a completion status of STATUS_ALERTED.

If the *WaitMode* parameter is *UserMode* and the *Alertable* parameter TRUE, then the thread can also be awakened to deliver a user mode APC. Kernel mode APCs always cause the subject thread to be awakened if the Wait IRQL is zero and no kernel APC is in progress.

The expiration time of the delay is expressed as either an absolute time at which the delay is to expire, or time that is relative to the current system time. If the value of the *Interval* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The value returned by **KeDelayExecutionThread** function determines how the delay was completed.

A value of STATUS_SUCCESS is returned if the delay was completed because the specified interval of time elapsed.

A value of STATUS_ALERTED is returned if the delay was completed because the thread was alerted.

A value of STATUS_USER_APC is returned if a user mode APC is to be delivered.

2.1.4.6 Disable Queuing of APCs

The queuing of APCs to a thread object can be disabled with the **KeDisableApcQueuingThread** function:

BOOLEAN
KeDisableApcQueuingThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Disabling the queuing of APCs to a thread object causes any attempt to direct an APC to the thread to be ignored.

During the termination of a thread, the executive must run down and clean up all thread data structures. When the APC queue itself is processed, the executive first disables APCs and then flushes the APC queue.

The previous value of the APC queuable state is returned as the function value. If the return value is TRUE, then APC queuing was previously enabled. Otherwise, a value of FALSE is returned and APC queuing was disabled.

2.1.4.7 Enable Queuing of APCs

The queuing of APCs to a thread object can be enabled with the **KeEnableApcQueuingThread** function:

```
BOOLEAN
KeEnableApcQueuingThread (
    IN PKTHREAD Thread
    );
```

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Enabling the queuing of APCs to a thread object allows APC objects to be inserted in the subject thread's APC queue for subsequent delivery when conditions permit.

The previous value of the APC queuable state is returned as the function value. If the return value is TRUE, then APC queuing was previously enabled. Otherwise, a value of FALSE is returned and APC queuing was disabled.

2.1.4.8 Force Resumption of Thread

A thread object's execution can be forced to resume with the **KeForceResumeThread** function:

```
ULONG
KeForceResumeThread (
    IN PKTHREAD Thread
    );
```

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Forcing the resumption of a thread object's execution checks the suspend and freeze count of the subject thread. If both counts are zero, then the thread is not currently suspended and no operation is performed. Otherwise, the subject thread's suspend and freeze counts are both set to zero, and the execution of the subject thread is resumed by releasing its builtin suspend semaphore.

The sum of the previous suspend and freeze counts is returned as the function value. If the return value is zero, then the subject thread was not previously suspended. Otherwise, the subject thread was suspended and its execution was resumed.

This function is intended for use by the executive when it wants to terminate the execution of a thread that may be in a suspended state.

2.1.4.9 Freeze Thread

The execution of a thread object can be frozen with the **KeFreezeThread** function:

ULONG
KeFreezeThread (
    IN PKTHREAD *Thread*
    );

<u>Parameters:</u>

Thread - A pointer to a dispatcher object of type thread.

Freezing a thread object causes the thread's freeze count to be incremented. If incrementing the freeze count would cause it to overflow, then the count is not incremented and the exception STATUS_SUSPEND_COUNT_EXCEEDED is raised. Otherwise, the freeze count is incremented. If the resultant value is one and the suspend count is zero, then the thread's builtin suspend APC object is queued.

When the suspend APC is delivered to the subject thread, a nonalertable Wait on the thread's builtin semaphore object is executed which freezes thread execution. The subject thread can be subsequently unfrozen with the **KeUnfreezeThread** function.

The previous freeze count is returned as the function value. If the return value is zero, then the subject thread was not previously frozen. Otherwise, the thread was previously frozen and must be unfrozen the number of times specified by the return value plus one before it will actually resume execution.

The freeze and unfreeze functions are similar to the suspend and resume functions, but are intended for use by system software as opposed to being exported to users. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.10 Query Data Alignment Mode

The data alignment handling mode for the current thread can be queried with the **KeQueryAutoAlignmentThread** function:

BOOLEAN
KeQueryAutoAlignmentThread (
)

The data alignment handling mode for the current thread is returned as the function value. A value of TRUE is returned if user mode data alignment exceptions are automatically handled by the kernel and are not raised as exceptions. Otherwise, user mode data alignment exceptions are not handled by the kernel and may, or may not, be raised as exceptions depending on host hardware capabilities. Automatic handling of user mode data alignment exceptions means that the kernel emulates misaligned data references and completes the offending instructions as if no misalignment exception had occurred. Misaligned references in kernel mode are never automatically handled and are always raised as exceptions.

IMPLEMENTATION NOTES:

Certain processors (e.g., the i386) always handle misaligned data in hardware. On these processors, enabling or disabling the automatic handling of data alignment exceptions has no effect. On other processors (e.g., i486, MIPS r3000, r4000SP, and r4000MP) the handling of misaligned data is handled according to the mode established for the respective thread.

2.1.4.11 Query Base Priority

The base priority of a thread object can be queried with the **KeQueryBasePriorityThread** function:

LONG
KeQueryBasePriorityThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

The base priority increment of the specified thread is returned as the function value. The base priority increment is defined as the difference between the specified thread's base priority and the base priority of the thread's process.

2.1.4.12 Read State Thread

The current state of a thread object can be read with the **KeReadStateThread** function:

BOOLEAN
KeReadStateThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

The current state of the thread object is returned as the function value. If the current state of the thread is Signaled, then a value of TRUE is returned. Otherwise, a value of FALSE is returned.

2.1.4.13 Ready Thread

A thread object can be readied for execution with the **KeReadyThread** function:

VOID
KeReadyThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Readying a thread object causes the thread to be considered for immediate execution on a processor.

If the thread's process is not in the balance set, then the thread's dispatching state is set to Ready and the thread object is inserted in the process' ready queue. Otherwise, an attempt is made to dispatch the thread on one of the processors in the multiprocessor complex. If the priority of the subject thread is greater than the priority of one or more threads running on processors that the subject thread can also run on, then the thread with the lowest priority is selected for preemption, the subject thread's dispatching state is set to Standby and an interprocessor interrupt is sent to the target processor to cause it to redispatch. Otherwise, the subject thread's dispatching state is set to Ready and the thread is inserted at the tail of the dispatcher ready queue selected by its priority.

2.1.4.14 Resume Thread

The execution of a thread object can be resumed with the **KeResumeThread** function:

ULONG
KeResumeThread (
    IN PKTHREAD *Thread*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Resuming a thread object checks the suspend count of the subject thread. If the suspend count is zero, then the thread is not currently suspended and no operation is performed. Otherwise, the subject thread's suspend count is decremented. If the resultant value is zero and the freeze count is also zero, then the execution of the subject thread is resumed by releasing its builtin suspend semaphore.

The previous suspend count is returned as the function value. If the return value is zero, then the subject thread was not previously suspended. If the return value is one, then the subject thread's execution was resumed. If the returned value is not zero or one, then the subject thread is still suspended and must be resumed the number of times specified by the return value minus one before it will actually resume execution.

The suspend and resume functions are similar to the freeze and unfreeze functions, but are usable from both system and user software. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.15 Rundown Thread

The current thread object can be run down with the **KeRundownThread** function:

VOID
KeRundownThread (
    );

This function runs down thread structures that are guarded by the dispatcher database lock and which must be processed before actually terminating the thread. An example of such a data structure is the mutant ownership list that is anchored in the thread object.

2.1.4.16 Set Affinity Thread

The affinity of a thread object can be set with the **KeSetAffinity** function:

KAFFINITY
KeSetAffinityThread (
    IN PKTHREAD *Thread*,
    IN KAFFINITY *Affinity*
    );

<u>Parameters:</u>

Thread - A pointer to a dispatcher object of type thread.

Affinity - The new set of processors on which the thread can run.

Setting the affinity of a thread object establishes a new set of processors on which the thread can execute. The new affinity must be a subset of the parent process's affinity. If the new affinity is zero, or is not a subset of the parent process's affinity, then an error condition is raised. Otherwise, the new affinity of the thread is set, and the previous affinity is returned as the function value.

If the dispatching state of the thread object is Running or Standby and the new affinity is such that the thread cannot execute on the target processor, then a new thread is selected for execution and an interprocessor interrupt is sent to the target processor.

2.1.4.17 Set Data Alignment Mode

The data alignment handling mode for the current thread can be set with the **KeSetAutoAlignmentThread** function:

BOOLEAN
KeSetAutoAlignmentThread (
    IN BOOLEAN *Enable*
)

<u>Parameters:</u>

Enable - A boolean variable that specifies the handling mode for data alignment exceptions in the current thread.

The *Enable* parameter specifies the handling mode for data alignment exceptions in the current thread. If this parameter is TRUE, then user mode data alignment exceptions are automatically handled by the kernel and are not raised as exceptions. Otherwise, user mode data alignment exceptions are not handled by the kernel and may, or may not, be raised as exceptions depending on host hardware capabilities. Automatic handling of user mode data alignment exceptions means that the kernel emulates misaligned data references and completes the offending instructions as if no misalignment exception had occurred. Misaligned references in kernel mode are never automatically handled and are always raised as exceptions.

The previous data alignment handling mode is returned as the function value.

IMPLEMENTATION NOTES:

Certain processors (e.g., the i386) always handle misaligned data in hardware. On these processors, enabling or disabling the automatic handling of data alignment exceptions has no effect. On other processors (e.g., i486, MIPS r3000, r4000SP, and r4000MP) the handling of misaligned data is handled according to the mode established for the respective thread.

2.1.4.18 Set Base Priority

The base priority of a thread object can be set with the **KeSetBasePriorityThread** function:

LONG
KeSetBasePriorityThread (
    IN PKTHREAD *Thread*,
    IN LONG *Increment*
    );

<u>Parameters:</u>

Thread - A pointer to a dispatcher object of type thread.

Increment - The base priority increment that is to be applied to the specified thread.

The new base priority is computed by adding the specified priority increment to the base priority of the specified thread's process. The resultant value is stored as the base priority of the specified thread.

The new base priority is restricted to the priority class of the specified thread's process. This means that the base priority is not allowed to cross over from a realtime priority class to a variable priority class or vice versa.

The previous base priority increment of the specified thread is returned as the function value. The previous base priority increment is defined as the difference between the specified thread's old base priority and the base priority of the thread's process.

2.1.4.19 Set Priority Thread

The priority of a thread object can be set with the **KeSetPriority** function:

KPRIORITY
KeSetPriorityThread (
    IN PKTHREAD *Thread*,
    IN KPRIORITY *Priority*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

Priority - The new priority of the thread.

Setting the priority of a thread object causes its eligibility for execution to be reexamined. The exact action that is taken depends on the current dispatching state of the thread. If the new priority is greater than the maximum thread priority, then an error condition is raised.

If the dispatching state of the thread object is Ready and the thread is currently inserted in the parent process's ready queue, then the priority of the thread is changed and no further action is taken.

If the dispatching state of the thread object is Ready and the thread is currently inserted in one of the dispatcher ready queues, then the thread is removed from its current ready queue, its priority is set to the specified value, and the thread is readied for execution as if it had just entered the ready state.

If the dispatching state of the thread object is Waiting or Terminated, then the priority of the thread is changed and no further action is taken.

If the dispatching state of the thread object is Standby or Running and the priority of the thread is being raised, then the priority of the thread is changed and no further action is taken.

If the dispatching state of the thread object is Standby or Running and the priority of the thread is being lowered, then a check is performed to determine if the thread should be preempted to run a higher priority thread. If a higher priority thread can execute on the target processor, then it is selected for execution and an interprocessor interrupt is sent to the target processor. Otherwise, no action is taken.

The previous priority of the subject thread is returned as the function value.

2.1.4.20 Suspend Thread

The execution of a thread object can be suspended with the **KeSuspendThread** function:

ULONG
KeSuspendThread (
  IN PKTHREAD *Thread*
  );

<u>Parameters:</u>

Thread - A pointer to a dispatcher object of type thread.

Suspending a thread object causes the thread's suspend count to be incremented. If incrementing the suspend count would cause it to overflow, then the count is not incremented and the exception STATUS_SUSPEND_COUNT_EXCEEDED is raised. Otherwise, the suspend count is incremented, and if the resultant value is one and the freeze count is zero, then the thread's builtin suspend APC object is queued.

When the suspend APC is delivered to the subject thread, a nonalertable Wait on the thread's builtin semaphore object is executed which suspends thread execution. The subject thread can be subsequently resumed with either the **KeResumeThread** or **KeAlertResumeThread** function.

The previous suspend count is returned as the function value. If the return value is zero, then the subject thread was not previously suspended. Otherwise, the thread was previously suspended and must be resumed the number of times specified by the return value plus one before it will actually resume execution.

The suspend and resume functions are similar to the freeze and unfreeze functions, but are usable from both system and user software. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.21 Terminate Thread

The execution of the current thread can be terminated with the **KeTerminateThread** function:

VOID
KeTerminateThread (
    IN KPRIORITY *Increment*
    );

Parameters:

Increment - The priority increment that is to be applied is terminating the current thread causes a Wait to be satisfied.

Terminating the current thread causes the dispatching state of the thread to be set to Terminated and the state of the thread object to be set to Signaled.

An attempt is made to satisfy as many Waits as possible for the current thread object.

A new thread object is selected for execution on the current processor and a context switch to the new thread is performed. There is no return from this function.

2.1.4.22 Test Alert Thread

An alert condition for the current thread can be tested for with the **KeTestAlertThread** function:

BOOLEAN
KeTestAlertThread (
    IN KPROCESSOR_MODE *AlertMode*
    );

Parameters:

AlertMode - The processor mode (*UserMode* or *KernelMode*) which is to be tested for an alert condition.

This function tests to determine if the current thread's alert variable for the specified processor mode has a value of TRUE or whether a user mode APC should be delivered to the current thread.

If the alert variable associated with the specified processor mode is TRUE, then it is set to a value of FALSE.

If the alert variable associated with the specified processor mode is FALSE and the specified processor mode is user, then the subject thread's APC queue is examined to determine whether a user mode APC should be delivered. If the user mode APC queue contains one or more entries, then the user APC pending variable is set to a value of TRUE in the current thread object.

The previous value of the alert variable for the specified processor mode is returned as the function value. If the return value is TRUE, then the current thread was alerted. Otherwise, a value of FALSE is returned and the current thread was not alerted.

2.1.4.23 Unfreeze Thread

The execution of a thread object can be unfrozen with the **KeUnfreezeThread** function:

ULONG
KeUnfreezeThread (
    IN PKTHREAD *Thread*
    );

<u>Parameters:</u>

> *Thread* - A pointer to a dispatcher object of type thread.

Unfreezing a thread object checks the freeze count of the subject thread. If the freeze count is zero, then the thread is not currently frozen and no operation is performed. Otherwise, the subject thread's freeze count is decremented. If the resultant value is zero and the suspend count is also zero, then the execution of the subject thread is unfrozen by releasing its builtin suspend semaphore.

The previous freeze count is returned as the function value. If the return value is zero, then the subject thread was not previously frozen. If the return value is one, then the subject thread's execution was unfrozen. If the returned value is not zero or one, then the subject thread is still frozen and must be unfrozen the number of times specified by the return value minus one before it will actually resume execution.

The freeze and unfreeze functions are similar to the suspend and resume functions, but are intended for use by system software as opposed to being exported to users. The freeze and unfreeze functions are used to suspend and resume thread execution during debugging operations.

2.1.4.24 Thread Performance Data

Several counters are maintained for each thread object to determine the various execution characteristics of the thread.

Two of the counters maintain the number of clock ticks that occurred while the thread was in user mode and while the thread was in kernel mode. These counters provide a quantitative measure of the distribution of computation time between the system and the user.

2.1.5 Timer Object

A *timer object* is used to record the passage of time. A timer object is set to a specified time and then expires when the time becomes due. When a timer object is set, its state is set to Not-Signaled and it is inserted in the system timer queue according to its expiration time. When the timer object expires, it is removed from the system timer queue and its state is set to Signaled.

A Deferred Procedure Call (DPC) can optionally be executed when a timer expires. This procedure executes at IRQL DISPATCH_LEVEL in the context of whatever thread happens to be executing when the timer expires.

Waiting on a timer object causes the execution of the subject thread to be suspended until the timer object attains a Signaled state. Satisfying the Wait for a timer object does not cause the state of the timer object to change. Therefore, when a timer object attains a Signaled state, an attempt is made to Satisfy as many Waits as possible.

Timer objects can be used to synchronize the execution of specific actions with time. This execution can occur at fixed points in time or at various intervals.

Programming interfaces that support the thread object include:

KeInitializeTimer - Initialize a timer object
KeCancelTimer - Cancel timer object expiration
KeReadStateTimer - Read state of timer object
KeSetTimer - Set timer object expiration time

2.1.5.1 Initialize Timer

A timer object can be initialized with the **KeInitializeTime** function:

VOID
KeInitializeTimer (
    IN PKTIMER *Timer*
    );

<u>Parameters:</u>

Timer - A pointer to a dispatcher object of type timer.

The timer object data structure is initialized with a state of Not-Signaled.

2.1.5.2 Cancel Timer

A timer object can be canceled with the **KeCancelTimer** function:

BOOLEAN
KeCancelTimer (
    IN PKTIMER *Timer*
    );

<u>Parameters:</u>

Timer - A pointer to a dispatcher object of type timer.

If the timer object is currently in the system timer queue, then it is removed from the queue and a value of TRUE is returned as the function value (a boolean state variable records whether the timer object is in the system timer queue). Otherwise, no operation is performed and a value of FALSE is returned as the function value.

2.1.5.3 Read State Timer

The current state of a timer object can be read with the **KeReadStateTimer** function:

BOOLEAN
KeReadStateTimer (
    IN PKTIMER *Timer*
    );

<u>Parameters:</u>

Timer - A pointer to a dispatcher object of type timer.

The current state of the timer object is returned as the function value. If the current state of the timer object is Signaled, then a value of TRUE is returned. Otherwise, a value of FALSE is returned.

2.1.5.4 Set Timer

A timer object can be set to expire at a specified time with the **KeSetTimer** function:

BOOLEAN
KeSetTimer (
    IN PKTIMER *Timer*,
    IN TIME *DueTime*,
    IN PKDPC *Dpc* **OPTIONAL**
    );

Parameters:

Timer - A pointer to a dispatcher object of type timer.

DueTime - The absolute or relative time at which the timer is to expire.

Dpc - An optional pointer to a control object of type deferred procedure call.

Setting a timer object causes the absolute expiration time to be computed, the state of the timer to be set to Not-Signaled, and the timer object to be inserted in the system timer queue. If the timer object is already in the timer queue, then it is implicitly canceled before it is set to the new expiration time (a boolean state variable records whether the timer object is in the system timer queue).

The expiration time of the timer object is expressed as either the absolute time that the timer is to expire, or a time that is relative to the current system time. If the value of the *DueTime* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The expiration time is expressed in system time units which are 100ns intervals.

If the *Dpc* parameter is specified, then a DPC object is associated with the timer object.

If the timer object was previously in the system timer queue, then a value of TRUE is returned as the function value. Otherwise, a value of FALSE is returned.

When the timer object expires, it is removed from the system timer queue and its state is set to Signaled. If a DPC object was associated with the timer object when it was set, then it is inserted in the system DPC queue and will execute as soon as conditions permit (a boolean state variable records whether the DPC object is in the system DPC queue).

2.2 Control Objects

This section describes the various types of control objects and the interfaces that are provided to manipulate these objects.

2.2.1 Asynchronous Procedure Call (APC) Object

An *Asynchronous Procedure Call* (APC) object provides the capability to break into the execution of a specified thread and cause a procedure to be called in a specified processor mode. Software running in kernel mode can only be interrupted to execute asynchronous procedures in kernel mode, whereas software running in user mode can be interrupted to execute asynchronous procedures in both user and kernel mode.

An asynchronous procedure call occurs in the context of a specified thread and is triggered by a software interrupt at APC_LEVEL.

There are two types of APC objects:

1. *Special*
2. *Normal*

Special APC objects cause the execution of a thread to be interrupted to execute a procedure in kernel mode at IRQL APC_LEVEL. Special APC objects can break into the execution of a thread at any time the thread is executing at IRQL 0.

Ordinarily, special APC procedures perform a minimal amount of work and return immediately to the APC dispatcher without calling any external procedures. However, if code running as part of a special APC procedure acquires a mutex that is also acquired by code outside the special APC procedure, then the code outside the special APC procedure must explicitly raise IRQL to APC_LEVEL when it wants to acquire the mutex.

This convention is required to prevent the special APC procedure from acquiring the mutex at an inappropriate time, which can happen if the thread that receives the special APC already owns the mutex when the special APC procedure is executed.

During the execution of a special APC procedure, page faults can be taken and all the kernel services are available. However, system services are not available, and care must be taken to ensure that any executive services that are used can be safely called.

Normal APC objects cause the execution of a thread to be interrupted to execute a procedure in kernel mode at IRQL APC_LEVEL and, in addition, a procedure in either kernel or user mode at IRQL 0. The first procedure (executed in kernel mode) is called just prior to calling the second procedure in the specified mode, and must adhere to the conventions for special APC procedures.

Normal APC objects can only break into the execution of a thread when the thread is executing at IRQL 0 and a normal APC for the specified mode is not already active.

While a normal APC is active for kernel mode, further normal APCs are software disabled until the active APC completes. The delivery of normal APCs for kernel mode is also implicitly disabled while a thread owns one or more mutexes (i.e, the thread does not have to explicitly raise IRQL to APC_LEVEL in order to synchronize with normal APC procedures).

Normal user mode APCs are only delivered when the subject thread is alertable. This occurs when a thread waits user-mode alertable and when the thread calls **KeTestAlertThread**.

While a normal APC is active for user mode, further normal APCs for user mode are software disabled by the alert mechanism. Upon completion of a normal APC in user mode, **KeTestAlertThread** is automatically called, which enables the delivery of another user mode APC. Thus, once a thread is user-mode alertable and an APC is delivered, further APCs are delivered one after the other until there are no APCs remaining in the user-mode APC queue.

During the execution of a normal APC procedure, all system operations are available and page faults can be taken.

Programming interfaces that support the APC object include:

> **KeInitializeApc** - Initialize an APC object
> **KeFlushQueueApc** - Flush all APC objects from APC queue
> **KeInsertQueueApc** - Insert APC object into APC queue
> **KeRemoveQueueApc** - Remove APC object from APC queue

2.2.1.1 Initialize APC

An APC object can be initialized with the **KeInitializeApc** function:

VOID
KeInitializeApc (
  IN PKAPC *Apc*,
  IN PKTHREAD *Thread*,
  IN KAPC_ENVIRONMENT *Environment*,
  IN PKKERNEL_ROUTINE *KernelRoutine*,
  IN PKRUNDOWN_ROUTINE *RundownRoutine* **OPTIONAL**,
  IN PKNORMAL_ROUTINE *NormalRoutine* **OPTIONAL**,
  IN KPROCESSOR_MODE *ApcMode* **OPTIONAL**,
  IN PVOID *NormalContext* **OPTIONAL**
  );

<u>Parameters:</u>

Apc - A pointer to a control object of type APC.

Thread - A pointer to a dispatcher object of type thread.

Environment - The environment in which the APC will execute (*OriginalApcEnvironment*, *AttachedApcEnvironment*, or *CurrentApcEnvironment*).

KernelRoutine - A pointer to a function that is to be executed at IRQL APC_LEVEL in kernel mode.

RundownRoutine - An optional pointer to a function that is to be executed if the APC object is contained in a thread's APC queue when the thread terminates.

NormalRoutine - An optional pointer to a function that is to be executed at IRQL 0 in the specified processor mode. If this parameter is not specified, then the *ApcMode* and *NormalContext* parameters are ignored.

ApcMode - The processor mode (*UserMode* or *KernelMode)* in which the function specified by the *NormalRoutine* parameter is to be executed. This parameter is ignored if the *NormalRoutine* parameter is not specified.

NormalContext - A pointer to an arbitrary data structure which is to be passed to the function specified by the *NormalRoutine* parameter. This parameter is ignored if the *NormalRoutine* parameter is not specified.

The function specified by the *KernelRoutine* parameter has the following type definition:

typedef
VOID
(***PKKERNEL_ROUTINE**) (
    IN PKAPC *Apc*,
    IN OUT PKNORMAL_ROUTINE **NormalRoutine*,
    IN OUT PVOID **NormalContext*,
    IN OUT PVOID **SystemArgument1*,
    IN OUT PVOID **SystemArgument2*
    );

<u>Parameters:</u>

Apc - A pointer to a control object of type APC.

NormalRoutine - A pointer to a pointer to the normal routine function that was specified when the APC was initialized.

NormalContext - A pointer to a pointer to an arbitrary data structure that was specified when the APC was initialized.

SystemArgument1, *SystemArgument2* - A set of two pointers to two arguments that contain untyped data.

The function specified by the *RundownRoutine* parameter has the following type definition:

typedef
VOID
(***PKRUNDOWN_ROUTINE**) (
    IN PKAPC *Apc*
    );

<u>Parameters:</u>

Apc - A pointer to a control object of type APC.

The function specified by the *NormalRoutine* parameter has the following type definition:

typedef
VOID
(***PKNORMAL_ROUTINE**) (
    IN PVOID *NormalContext*,
    IN PVOID *SystemArgument1*,
    IN PVOID *SystemArgument2*
    );

Parameters:

NormalContext - A pointer to an arbitrary data structure that was specified when the corresponding APC object was initialized.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

The type of APC object to be initialized is determined by the presence or absence of the optional *NormalRoutine* parameter. If the *NormalRoutine* parameter is present, then a normal APC object is initialized and the values of the *ApcMode* and *NormalContext* parameters are stored in the APC object. Otherwise, a special APC object is initialized for execution in kernel mode.

The *Environment* parameter specifies the execution environment of the specified APC object. An APC object can be executed in the context of a thread's parent process or a process to which the thread has attached.

The *KernelRoutine* parameter specifies the procedure that is to be called in kernel mode at IRQL APC_LEVEL. This procedure is called with a copy of the parameters that are specified for the normal routine when the APC is initialized and can modify these parameters as necessary to alter the execution of the normal routine. If the normal routine is not specified when the APC is initialized, then any assignment to these parameters is ignored.

If specified, the *RundownRoutine* parameter specifies a procedure that is to be called when a thread terminates with the APC object in its APC queue. The purpose of this routine is to allow for special disposition of the APC object during thread rundown.

If specified, the *NormalRoutine* parameter specifies the procedure that is to be executed in the processor mode specified by the *ApcMode* parameter. This procedure will be called with the *NormalContext* parameter and two additional arguments provided by the system when the APC queued.

In order to actually interrupt the execution of a thread, an APC object must be inserted into one of the thread's APC queues (there is a separate APC queue for user and kernel mode).

2.2.1.2 Flush Queue APC

All the APC objects in a specified thread's APC queue can be flushed with the **KeFlushQueueApc** function:

PLIST_ENTRY
KeFlushQueueApc (
    IN PKTHREAD *Thread*,
    IN KPROCESSOR_MODE *ApcMode*
    );

Parameters:

Thread - A pointer to a dispatcher object of type thread.

ApcMode - The processor mode (*UserMode* or *KernelMode*) of the APC queue that is to be flushed.

An APC queue is flushed by removing the APC listhead from the list of APC entries, reinitializing the APC listhead, and returning the list of APC objects as the function value. If the APC queue is empty, then a NULL pointer is returned. Otherwise, the address of the list entry for the first APC object is returned as the function value. It is the responsibility of the caller to scan the list of APC objects and dispense with each object as appropriate.

The APC objects are linked together by a list entry in each object. Scanning this list can be accomplished using the CONTAINING_RECORD function to locate the address of the respective APC object given the address of its list entry.

This function is used by the executive during thread termination to remove remaining entries from the thread's APC lists.

2.2.1.3 Insert Queue APC

An APC object can be inserted into a thread's APC queue with the **KeInsertQueueApc** function:

BOOLEAN
KeInsertQueueApc (
    IN PKAPC *Apc*,
    IN PVOID *SystemArgument1*,
    IN PVOID *SystemArgument2*,
    IN KPRIORITY *Increment*
    );

Parameters:

Apc - A pointer to a control object of type APC.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

Increment - The priority increment that is to be applied if queuing the APC causes the target thread's wait to be satisfied.

If the specified APC object is already in an APC queue (a boolean state variable records whether the APC object is in an APC queue) or APC queuing is disabled for the subject thread, then no operation is performed and a function value of FALSE is returned. Otherwise, the APC object is inserted into the APC queue specified by the *ApcMode* and *Thread* parameters that were supplied when the APC object was initialized and a function value of TRUE is returned.

When proper enabling conditions are present, the APC will be delivered to the subject thread and the specified procedure(s) will be executed in the specified processor mode.

A special APC is deliverable whenever the IRQL of the subject thread is zero.

A normal kernel APC is deliverable whenever the IRQL of the subject thread is zero, a normal kernel APC is not already in progress, and the subject thread does not own any mutexes.

A normal user APC is deliverable when the subject thread waits user-mode alertable and when the subject thread calls **KeTestAlertThread**.

2.2.1.4 Remove Queue APC

An APC object can be removed from an APC queue with the **KeRemoveQueueApc** function:

BOOLEAN
KeRemoveQueueApc (
    IN PKAPC *Apc*
    );

Parameters:

Apc - A pointer to a control object of type APC.

If the specified APC object is not currently in an APC queue (a boolean state variable records whether the APC object is in an APC queue), then a value of FALSE is returned and no operation is performed. Otherwise, the specified APC object is removed from its APC queue and a function value of TRUE is returned.

2.2.2 Deferred Procedure Call (DPC) Object

A *Deferred Procedure Call* (DPC) object provides the capability to break into the execution of the current thread and cause a procedure to be executed in kernel mode at IRQL DISPATCH_LEVEL.

There is one DPC queue for the entire system. When a DPC object is inserted in the DPC queue, a software interrupt is requested at DISPATCH_LEVEL on the current processor. As soon as the IRQL falls below DISPATCH_LEVEL, a software interrupt will be taken which will cause the DPC dispatcher to execute.

The DPC dispatcher removes entries from the DPC queue, calls the specified procedure, and upon return, removes another entry from the queue. This is continued until there are no longer any entries in the DPC queue, at which time the DPC dispatcher checks to determine if another thread has been selected for execution on the current processor. If a thread has been selected, then a context switch to that thread is performed. Otherwise, the interrupt is dismissed and execution of the current thread is continued.

A deferred procedure call occurs at IRQL DISPATCH_LEVEL in the context of whatever thread was interrupted when the DISPATCH_LEVEL interrupt occurred. A very limited set of operations can be performed by the DPC procedure.

No system services can be executed by the DPC procedure nor can any page faults be taken. The kernel services are generally available. However, the Wait functions can only be called if it is known that they will not actually cause a wait to occur. (Using an explicit time-out value of zero implements a conditional Wait operation). If page faults or waits were allowed, then it would be possible to randomly cause an arbitrary thread to wait in the kernel at IRQL DISPATCH_LEVEL causing possible deadlocks and data corruption.

Deferred procedure execution is intended mainly for use by device drivers that need to lower their IRQL to complete an I/O operation. The kernel itself, however, uses DPC objects to implement timers, quantum end, and power failure recovery.

Programming interfaces that support the DPC object include:

> **KeInitializeDpc -** Initialize a DPC object
> **KeInsertQueueDpc** - Insert DPC object into the DPC queue
> **KeRemoveQueueDpc** - Remove DPC object from the DPC queue

2.2.2.1 Initialize DPC

A DPC object can be initialized with the **KeInitializeDpc** function:

VOID
KeInitializeDpc (
    IN PKDPC *Dpc*,
    IN PKDEFERRED_ROUTINE *DeferredRoutine*,
    IN PVOID *DeferredContext*
    );

<u>Parameters:</u>

> *Dpc* - A pointer to a control object of type DPC.
>
> *DeferredRoutine* - A pointer to a function that is to be called when the DPC object is removed from the DPC queue.
>
> *DeferredContext* - A pointer to an arbitrary data structure that is to be passed to the function specified by the *DeferredRoutine* parameter.

The function specified by the *DeferredRoutine* parameter has the following type definition:

```
typedef
VOID
(*PKDEFERRED_ROUTINE) (
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
    );
```

Parameters:

Dpc - A pointer to a control object of type DPC.

DeferredContext - A pointer to an arbitrary data structure that was specified when the DPC was initialized.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

2.2.2.2 Insert Queue DPC

A DPC object can be inserted in the system DPC queue with the **KeInsertQueueDpc** function:

```
BOOLEAN
KeInsertQueueDpc (
    IN PKDPC Dpc,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
    );
```

Parameters:

Dpc - A pointer to a control object of type DPC.

SystemArgument1, *SystemArgument2* - A set of two arguments that contain untyped data.

If the specified DPC object is already in the DPC queue (a boolean state variable records whether the DPC object is in the DPC queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the DPC object is inserted in the DPC queue, a software interrupt is request at IRQL DISPATCH_LEVEL on the current processor, and a function value of TRUE is returned.

The deferred procedure will be executed as soon as the IRQL of the current processor drops below DISPATCH_LEVEL.

2.2.2.3 Remove Queue DPC

A DPC object can be removed from the DPC queue with the **KeRemoveQueueDpc** function:

BOOLEAN
KeRemoveQueueDpc (
    IN PKDPC *Dpc*
    );

Parameters:

Dpc - A pointer to a control object of type DPC.

If the specified DPC object is not currently in the DPC queue (a boolean state variable records whether the DPC object is in the DPC queue), then a value of FALSE is returned and no operation is performed. Otherwise, the specified DPC object is removed from the DPC queue and a function value of TRUE is returned.

2.2.3 Device Queue Object

A *device queue object* is used to record the state of a device driver and to provide a queue into which I/O requests can be placed for subsequent processing.

A device queue object has a state which is either *Busy* or *Not-Busy*.

When the state of a device queue object is Not-Busy, the associated device driver is idle and therefore not performing any work.

A device queue object transitions to the Busy state when an attempt is made to insert a device queue entry into a device queue that is empty. For this case, the device queue entry is not actually placed in the device queue, but rather, the device queue object is marked Busy and a boolean value of FALSE is returned to signify that the associated device driver should process the device queue entry immediately.

Once a device queue object is Busy, further I/O requests are placed in the device queue in either a FIFO or key-sorted order.

A device queue object transitions to a Not-Busy state when an attempt is made to remove a device queue entry from a device queue object and the corresponding device queue is empty.

A device queue entry has the following type definition:

typedef struct _KDEVICE_QUEUE_ENTRY {
    LIST_ENTRY *DeviceListEntry*;
    ULONG *SortKey*;
    BOOLEAN *Inserted*;
} **KDEVICE_QUEUE_ENTRY**;

Programming interfaces that support the device queue object include:

KeInitializeDeviceQueue - Initialize a device queue
KeInsertDeviceQueue - Insert entry at tail of device queue
KeInsertByKeyDeviceQueue - Insert entry by key into device queue
KeRemoveDeviceQueue - Remove entry from head of device queue
KeRemoveEntryDeviceQueue - Remove entry from device queue

2.2.3.1 Initialize Device Queue

A device queue object can be initialized with the **KeInitializeDeviceQueue** function:

VOID
KeInitializeDeviceQueue (
    IN PKDEVICE_QUEUE *DeviceQueue*,
    IN PKSPIN_LOCK *SpinLock*
    );

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

SpinLock - A pointer to an executive spin lock.

The device queue object data structure is initialized and the state of the device queue is set to Not-Busy.

2.2.3.2 Insert Device Queue

An entry can be inserted at the tail of a device queue with the **KeInsertDeviceQueue** function:

BOOLEAN
KeInsertDeviceQueue (
    IN PKDEVICE_QUEUE *DeviceQueue*,
    IN PKDEVICE_QUEUE_ENTRY *DeviceQueueEntry*
    );

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

DeviceQueueEntry - A pointer to the device queue entry that is to be inserted at the tail of the device queue.

The specified device queue spin lock is acquired, and the state of the device queue is checked.

If the state of the device queue is Not-Busy, then the state of the device queue is set to Busy, the device queue spin lock is released, and a value of FALSE is returned as the function value (i.e., the device queue entry is not inserted in the device queue).

If the state of the device queue is Busy, then the specified device queue entry is inserted at the tail of device queue, the device queue spin lock is released, and a value of TRUE is returned as the function value.

This function is intended for use by code that queues an I/O request to a device driver. It must be called from an IRQL of DISPATCH_LEVEL.

2.2.3.3 Insert By Key Device Queue

An entry can be inserted into a device queue according to a key value with the **KeInsertByKeyDeviceQueue** function:

BOOLEAN
KeInsertByKeyDeviceQueue (
    IN PKDEVICE_QUEUE *DeviceQueue*,
    IN PKDEVICE_QUEUE_ENTRY *DeviceQueueEntry*,
    IN ULONG *SortKey*
    );

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

DeviceQueueEntry - A pointer to the device queue entry that is to be inserted into the device queue by key.

SortKey - The sort key value that is to be used to determine the position at which the device queue entry is to be inserted in the specified device queue.

The specified device queue spin lock is acquired, and the state of the device queue is checked.

If the state of the device queue is Not-Busy, then the state of the device queue is set to Busy, the device queue spin lock is released, and a value of FALSE is returned as the function value (i.e., the device queue entry is not inserted in the device queue).

If the state of the device queue is Busy, then the specified device queue entry is inserted into the device queue according to its sort key value, the device queue spin lock is released, and a value of TRUE is returned as the function value.

Insertion in the device queue is such that the preceding entry in the queue has a sort key that is less than or equal to the new entry's sort key and the succeeding entry has a sort key that is greater than the new entry's sort key.

This function is intended for use by code that queues an I/O request to a device driver. It must be called from an IRQL of DISPATCH_LEVEL.

2.2.3.4 Remove Device Queue

An entry can be removed from the head of a device queue with the **KeRemoveDeviceQueue** function:

```
PKDEVICE_QUEUE_ENTRY
KeRemoveDeviceQueue (
    IN PKDEVICE_QUEUE DeviceQueue
    );
```

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

This function can only be called from an IRQL of DISPATCH_LEVEL and is intended for use by device driver code that completes one I/O request and starts the next one.

The specified device queue spin lock is acquired and the state of the device queue is checked.

If the state of the device queue is Not-Busy, then a bug check will occur (i.e., **KeRemoveDeviceQueue** cannot be called when the device queue is Not-Busy).

If the state of the device queue is Busy, then an attempt is made to remove an entry from the head of the device queue. If the device queue is empty, then the state of the device queue is set to Not-Busy and a NULL pointer is returned as the function value. Otherwise, the next entry is removed from the head of the device queue, the inserted status of the entry is set to FALSE, and the address of the entry is returned as the function value.

The specified device queue spin lock is released.

2.2.3.5 Remove Entry Device Queue

A specific entry can be removed from a device queue with the **KeRemoveEntryDeviceQueue** function:

BOOLEAN
KeRemoveEntryDeviceQueue (
    IN PKDEVICE_QUEUE *DeviceQueue,*
    IN PKDEVICE_QUEUE_ENTRY *DeviceQueueEntry*
    );

Parameters:

DeviceQueue - A pointer to a control object of type device queue.

DeviceQueueEntry - A pointer to the device queue entry that is to be removed from the specified device queue.

The IRQL is raised to DISPATCH_LEVEL and the specified device queue spin lock is acquired.

If the specified device queue entry is currently in a device queue (a boolean state variable records whether a device queue entry is in a device queue), then the device queue entry is removed from the device queue, the inserted status of the device queue entry is set to FALSE, and a value of TRUE is returned as the function value. Otherwise, the specified device queue entry is not in a device queue and a value of FALSE is returned.

The specified device queue spin lock is released and IRQL is restored to its previous value.

This function is intended for use in canceling I/O operations and is callable from any IRQL that is less than or equal to DISPATCH_LEVEL.

2.2.4 Interrupt Object

An *interrupt object* provides the capability to connect an interrupt source to an interrupt service routine via an entry in an Interrupt Dispatch Table (IDT). Each processor has an IDT that is used to dispatch interrupts which occur on that processor.

The IDT is a software-defined table that contains an entry for each of the Interrupt Request Levels (IRQLs). When an interrupt occurs at one of these levels, the interrupt dispatcher reads the IRQL of the interrupting source from the interrupt controller. This value is then used to locate the corresponding entry in the IDT that is used to dispatch the execution of the associated service routine.

Several of the IDT entries are reserved for use by the kernel and cannot be connected to interrupt objects. These entries include the following:

- PASSIVE_LEVEL - Passive release
- APC_LEVEL - Asynchronous Procedure Call
- DISPATCH_LEVEL - Dispatch and Deferred Procedure Call
- WAKE_LEVEL - Wake system debugger
- CLOCK2_LEVEL - Interval timer
- IPI_LEVEL - Interprocessor request
- POWER_LEVEL - Power failure
- HIGH_LEVEL - Machine check

The remaining levels can be used for device interrupts or bus adapters.

In addition to the 16 entries that are directly associated with the hardware interrupt request levels, there are 48 more entries in the IDT that are provided to allow secondary level dispatching of interrupts. These entries can be used by a first-level service routine (i.e., one connected to IRQLs 0 - 15) to dispatch secondary level interrupts such as those that might be received from a bus adapter. For example, a bus adapter might have several devices that can cause interrupts and a mechanism for identifying which device is requesting service. When a bus adapter interrupt is received, the service routine connected to the appropriate first level IDT entry is executed. This service routine reads the adapter register that identifies the interrupting device and uses the information to locate the appropriate second-level

IDT entry. The second-level IDT entry must be connected to an interrupt object and contains the address of the interrupt transfer routine which is called.

An interrupt transfer routine has the following type definition:

typedef
BOOLEAN
(*PKTRANSFER_ROUTINE) (
    );

Interrupt sources are classified as either *LevelSensitive* or *Latched*. Level sensitive interrupts request an interrupt whenever the corresponding interrupt request signal is asserted. The service routine associated with the interrupt source must remove the cause of the interrupt before the interrupt request is dropped. Latched interrupts are requested whenever the corresponding interrupt request signal transitions from the deasserted to the asserted state.

An interrupt object can only be connected to a single IDT entry. If a particular service routine must be connected to the same interrupt on multiple processors, then multiple interrupt objects must be used. Multiple interrupt objects can be connected to a single IDT entry. They must, however, all have the same interrupt type (i.e., level sensitive or latched).

Interrupt objects are intended for use by device drivers.

> \ *Kernel code that utilizes interrupts directly does not connect interrupts using this object. These interrupts include the interval timer, power failure, machine check, and the two software interrupt levels. The code for these interrupts is written in assembler since it is small and system dependent.* \

Programming interfaces that support the interrupt object include:

KeInitializeInterrupt - Initialize an interrupt object
KeConnectInterrupt - Connect interrupt object to an IDT entry
KeDisconnectInterrupt - Disconnect interrupt object from an IDT entry
KeSynchronizeExecution - Synchronize execution with an interrupt

2.2.4.1 Initialize Interrupt

An interrupt object can be initialized with the **KeInitializeInterrupt** function:

VOID
KeInitializeInterrupt (
    IN PKINTERRUPT *Interrupt*,
    IN PKSERVICE_ROUTINE *ServiceRoutine*,
    IN PVOID *ServiceContext*,
    IN PKSPIN_LOCK *SpinLock*,
    IN CCHAR *Vector*,
    IN KIRQL *InterruptIrql*,
    IN KIRQL *SynchronizeIrql*,
    IN KINTERRUPT_MODE *InterruptMode*,
    IN BOOLEAN *ShareVector*,
    IN CCHAR *ProcessorNumber*,
    IN BOOLEAN *FloatingSave*
    );

Parameters:

Interrupt - A pointer to a control object of type interrupt.

ServiceRoutine - A pointer to a function that is to be called when an interrupt occurs on the specified processor through the specified IDT entry.

ServiceContext - A pointer to an arbitrary data structure which will be passed to the *ServiceRoutine* function as a parameter.

SpinLock - A pointer to an spin lock that is to be used to synchronize the execution of the *ServiceRoutine* function with the corresponding device driver.

Vector - The index of the entry in the specified IDT that is to be associated with *ServiceRoutine* function.

InterruptIrql - The request priority of the interrupting source.

SynchronizeIrql - The request priority that the interrupt should be synchronzied with.

InterruptMode - The mode of the interrupt (*LevelSensitive* or *Latched*).

ShareVector - A boolean that specifies whether the interrupt vector to which the object is connected may be shared. If FALSE then the vector may not be shared, if TRUE it may be.

ProcessorNumber - The number of the processor whose IDT is to used when connecting the interrupt.

FloatingSave - A boolean variable that specifies whether the floating point context needs to be saved when a interrupt is received from the interrupt source.

The function specified by the *ServiceRoutine* parameter has the following type definition:

typedef
BOOLEAN
(***PKSERVICE_ROUTINE**) (
    IN PKINTERRUPT *Interrupt*,
    IN PVOID *ServiceContext*
    );

<u>Parameters:</u>

Interrupt - A pointer to a control object of type interrupt which is connected to the associated interrupt source.

ServiceContext - A pointer to an arbitrary data structure that was specified when the corresponding interrupt object was initialized.

The interrupt object is initialized with the specified parameters. In order for the function specified by the *ServiceRoutine* parameter to actually get called when an interrupt is received from the interrupt source, the interrupt object must be connected to the specified IDT entry using the **KeConnectInterrupt** function.

The spin lock specified by the *SpinLock* parameter is used to synchronize execution.

If *SynchronizeIrql* is not equal to *InterruptIrql*, then the system will raise its priority level to *SynchronizeIrql* level before acquiring the lock specified by *SpinLock*. This allows support for devices with multiple interrupt sources, since all can be synchronized with a single spin lock at a single priority level.

It is an error for *SynchronizeIrql* to be less than *InterruptIrql*, the system will refuse to connect such an interrupt object.

An interrupt object can only be connected to a single IDT entry on a single processor. The *Vector* parameter specifies the IDT entry and the *ProcessorNumber* parameter specifies which IDT is to be used. If a particular service routine must be connected to the same IDT entry on multiple processors, then multiple interrupt objects must be used. More than one interrupt object, however, can be connected to the same IDT entry on the same processor, but all such interrupt objects must have been initialized with *ShareVector* set to TRUE. When this happens, appropriate data structures are automatically set up to call each connected interrupt service routine one after the other.

The mode of the interrupt specifies whether the interrupt is a *LevelSensitive* or *Latched* interrupt. Level sensitive interrupts are continually requested as long as the interrupt signal stays asserted. Therefore, the interrupt service routine must remove the reason for the interrupt before returning control. Latched interrupts are requested only on the transition of the interrupt signal from deasserted to asserted. The function specified by the *ServiceRoutine* parameter must return a boolean value that signifies whether the interrupt was handled or not.

The *ShareVector* parameter declares whether the interrupt object may be connected to its interrupt vector at the same time as other interrupt objects. All interrupt objects sharing an interrupt vector must have *ShareVector* set to TRUE. The system may disallow sharing of an interrupt vector, even if all interrupt objects for which connections are attempted have *ShareVector* set to TRUE. This could happen because the underlying hardware does not support sharing.

The *FloatingSave* parameter specifies whether the *ServiceRoutine* function uses the floating point registers. If this parameter is TRUE, then the floating context is saved before calling the specified service routine. Otherwise, it is not saved and a fair amount of overhead is saved.

> \ *This parameter is being provided with the hope that a compiler option will be implemented that allows a module to be compiled such that it will not use the floating point registers. This option does not currently exist and this parameter should always be specified as TRUE.* \

Initializing an interrupt causes code to be generated that will synchronize execution with the appropriate interrupt object, call the specified interrupt service routine, and dismiss the interrupt.

2.2.4.2 Connect Interrupt

An interrupt object can be connected to an IDT entry with the **KeConnectInterrupt** function:

BOOLEAN
KeConnectInterrupt (
    IN PKINTERRUPT *Interrupt*
    );

<u>Parameters:</u>

> *Interrupt* - A pointer to a control object of type interrupt.

If the specified interrupt object is already connected (a boolean state variable records whether the interrupt object is connected), the specified vector number is greater than the maximum vector, the specified IRQL is greater than HIGH_LEVEL, the specified level cannot be connected to (e.g., a reserved level, sharing conflicts), or the specified processor number is greater than the number of processors in the configuration, then no operation is performed and a function value of FALSE is returned. Otherwise, the interrupt object is connected to the IDT entry that was specified when the interrupt object was initialized and a function value of TRUE is returned.

Once an interrupt object is connected to an IDT entry, the corresponding service routine will be called each time an interrupt is received from that interrupt source. If multiple interrupt objects are connected to a single IDT entry, then the service routines are called in the order in which they were connected.

2.2.4.3 Disconnect Interrupt

An interrupt object can be disconnected from an IDT entry with the **KeDisconnectInterrupt** function:

```
BOOLEAN
KeDisconnectInterrupt (
    IN PKINTERRUPT interrupt
    );
```

Parameters:

Interrupt - A pointer to a control object of type interrupt.

If the specified interrupt object is not connected (a boolean state variable records whether the interrupt object is connected), then no operation is performed and a function value of FALSE is returned. Otherwise, the interrupt object is disconnected from the IDT entry that was specified when the interrupt object was initialized and a function value of TRUE is returned.

Further interrupts received from the interrupting source will be logged, but otherwise ignored.

2.2.4.4 Synchronize Execution

The execution of a device driver function can be synchronized with the execution of the service routine associated with an interrupt object with the **KeSynchronizeExecution** function:

BOOLEAN
KeSynchronizeExecution (
    IN PKINTERRUPT *Interrupt*,
    IN PKSYNCHRONIZE_ROUTINE *SynchronizeRoutine*,
    IN PVOID *SynchronizeContext*
    );

Parameters:

Interrupt - A pointer to a control object of type interrupt.

SynchronizeRoutine - A pointer to a device driver function whose execution is to be synchronized with the execution of the service routine associated with the specified interrupt object.

SynchronizeContext - A pointer to an arbitrary data structure which is to be passed to the function specified by the *SynchronizeRoutine* parameter.

The function specified by the *SynchronizeRoutine* parameter has the following type definition:

typedef
BOOLEAN
(***PKSYNCHRONIZE_ROUTINE**) (
    IN PVOID *SynchronizeContext*
    );

Parameters:

ServiceContext - A pointer to an arbitrary data structure that was specified when the call to **KeSynchronizeExecution** was executed.

This function is used by a device driver to synchronize execution with a service routine which may be executing on another processor in a multiprocessor configuration. Such synchronization is only necessary in those cases where both the service routine and device driver access the same resources in a way that requires mutually exclusive access.

When this function is executed, the IRQL is raised to the level specified by the interrupt source's interrupt object (the higher of *InterruptIrql* and *SynchronizeIrql*), access is synchronized with the corresponding service routine by acquiring the associated spin lock, and then the specified routine is called. The routine should access resources as necessary and return a boolean value. Upon return, the IRQL is restored and the boolean value is returned as the function value.

Routines executed with this function execute at an elevated IRQL and must be very short in duration. It is intended that these routines be used for such purposes as loading device registers and should be only a few microseconds in length.

The boolean return value is intended to be attached to the occurrence of a power failure. A device driver can use a power status object to record the occurrence of a power failure. The synchronize routine should raise IRQL to POWER_LEVEL and test the corresponding status variable before loading any device registers. If the value is TRUE, then power has failed and the device may not be in an appropriate state. If the value is FALSE, then power has not failed and a sequence of device register loads can be performed without a power failure since power failure interrupts are disabled.

2.2.5 Power Notify Object

A *power notify object* provides the capability to automatically have a specified function called when power is restored after a power failure.

This object is intended for use by device drivers and other code that needs to be asynchronously notified via a function call when power is restored after a failure. The function call can be used to reinitialize device state, restart I/O operations, etc.

A power notify object, when inserted in the power notify queue, is a repeatable operation. That is, the specified function will be called each time the power is restored. The kernel guarantees that once called, the specified function will not be recursively recalled until it has completed its execution and returned to the kernel.

When power is restored after a power failure, the kernel scans the power notify queue and calls the specified functions in the order in which they were inserted. Thus layered device drivers can ensure that they are called in the correct order.

If the power fails and is restored during the scan of the power notify queue, then the scan is immediately restarted at the beginning of the queue.

A power notify object cannot be inserted in, or removed from, the power notify queue from a function that is called as the result of power restoration (i.e., a power notify routine).

Programming interfaces that support the power notify object include:

KeInitializePowerNotify - Initialize notify object
KeInsertQueuePowerNotify - Insert power notify object
KeRemoveQueuePowerNotify - Remove power notify object

2.2.5.1 Initialize Power Notify

A power notify object can be initialized with the **KeInitializePowerNotify** function:

VOID
KeInitializePowerNotify (
    IN PKPOWER_NOTIFY *PowerNotify*,
    IN PKNOTIFY_ROUTINE *NotifyRoutine*,
    IN PVOID *NotifyContext*
    );

Parameters:

PowerNotify - A pointer to a control object of type power notify.

NotifyRoutine - A pointer to a function that is to be called when power is restored after a power failure.

NotifyContext - A pointer to an arbitrary data structure which will be passed to the *NotifyRoutine* as a parameter.

The function specified by the *NotifyRoutine* parameter has the following type definition:

typedef
VOID
(*PKNOTIFY_ROUTINE) (
    IN PVOID *NotifyContext*
    );

Parameters:

NotifyContext - A pointer to an arbitrary data structure that was specified when the power notify object was initialized.

The power notify object data structure is initialized.

In order to actually have the specified function called when power is restored after a failure, the power notify object must be inserted in the power notify queue.

2.2.5.2 Insert Power Notify

A power notify object can be inserted in the power notify queue with the **KeInsertQueuePowerNotify** function:

BOOLEAN
KeInsertQueuePowerNotify (
    IN PKPOWER_NOTIFY *PowerNotify*
    );

<u>Parameters:</u>

PowerNotify - A pointer to a control object of type power notify.

If the specified power notify object is already in the power notify queue (a boolean state variable records whether the power notify object is in the power notify queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power notify object is inserted in the power notify queue and a function value of TRUE is returned.

When the power is restored after a failure, the kernel scans the power notify queue and calls the specified function.

2.2.5.3 Remove Power Notify

A power notify object can be removed from the power notify queue with the **KeRemoveQueuePowerNotify** function:

BOOLEAN
KeRemoveQueuePowerNotify (
    IN PKPOWER_NOTIFY *PowerNotify*
    );

<u>Parameters:</u>

PowerNotify - A pointer to a control object of type power notify.

If the power notify object is not in the power notify queue (a boolean state variable records whether the power notify object is in the power notify queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power notify object is removed from the power notify queue and a function value of TRUE is returned.

2.2.6 Power Status Object

A *power status object* provides the capability to automatically have a boolean state variable set to a value of TRUE when power is restored after a power failure.

This object is intended for use by device drivers and other code which needs to synchronize access to volatile register and device state such that a power failure does not leave the registers or device in an indeterminate state. The boolean value can be interrogated at critical points during driver execution to determine whether a given operation should be continued or restarted.

A power status object, when inserted in the power status queue, is a one-shot operation. That is, the boolean variable will be set to a value of TRUE exactly once after the power is restored. If it is desirable to have the boolean variable set to a value of TRUE the next time that power fails, then the power status object must be reinserted in the power status queue.

Programming interfaces that support the power status object include:

> **KeInitializePowerStatus** - Initialize status object
> **KeInsertQueuePowerStatus** - Insert power status object
> **KeRemoveQueuePowerStatus** - Remove power status object

2.2.6.1 Initialize Power Status

A power status object can be initialized with the **KeInitializePowerStatus** function:

VOID
KeInitializePowerStatus (
    IN PKPOWER_STATUS *PowerStatus*
    );

Parameters:

> *PowerStatus* - A pointer to a control object of type power status.

The power status object data structure is initialized.

In order to actually have a boolean variable set to a value of TRUE when power is restored after a failure, the power status object must be inserted in the power status queue.

2.2.6.2 Insert Power Status

A power status object can be inserted in the power status queue with the **KeInsertQueuePowerStatus** function:

```
BOOLEAN
KeInsertQueuePowerStatus (
    IN PKPOWER_STATUS PowerStatus,
    IN PBOOLEAN Status
    );
```

Parameters:

PowerStatus - A pointer to a control object of type power status.

Status - A pointer to a boolean variable that is to be set to a value of TRUE when power is restored after a failure.

If the specified power status object is already in the power status queue (a boolean state variable records whether the power status object is in the power status queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power status object is inserted in the power status queue, the specified boolean variable is set to a value of FALSE, and a function value of TRUE is returned.

When the power is restored after a failure, the kernel removes each entry from the power status queue, sets the specified boolean variable to a value of TRUE, and sets the inserted state of the power status object to FALSE.

2.2.6.3 Remove Power Status

A power status object can be removed from the power status queue with the **KeRemoveQueuePowerStatus** function:

```
BOOLEAN
KeRemoveQueuePowerStatus (
    IN PKPOWER_STATUS PowerStatus
    );
```

Parameters:

PowerStatus - A pointer to a control object of type power status.

If the power status object is not in the power status queue (a boolean state variable records whether the power status object is in the power status queue), then no operation is performed and a function value of FALSE is returned. Otherwise, the power status object is removed from the power status queue and a function value of TRUE is returned.

2.2.7 Process Object

A *process object* represents the virtual address space and control information necessary for the execution of a set of thread objects.

A process object contains a pointer to an address map, a thread ready list to hold thread objects while the process is not in the balance set, a list of threads that are children of the process, the total accumulated time for all threads executing within the process, a base priority, and a default thread affinity.

A process object must be initialized before any thread objects can be initialized that specify the process as their parent.

A process is either in the balance set (Included) or not in the balance set (Excluded). When a process is in the balance set, then all threads that are children of the process are eligible to be considered for execution on a processor. When a process is not in the balance set, then necessary pages are not locked in memory (e.g. thread kernel stacks) and threads that are children of the process are not eligible for execution on a processor.

The balance set is managed by the balance set manager; see also the discussion under Thread Object.

A process cannot leave the balance set while any of its children threads own mutexes. Therefore, when a process is selected for removal from the balance set, any children threads that own mutexes are allowed to continue execution until they release their last mutex. When this occurs, execution of the thread is suspended and it is placed in the process ready queue rather than returning to one of the dispatcher ready queues. When no threads in the process own mutexes, then the process can actually be removed from the balance set.

Programming interfaces that support the process object include:

KeInitializeProcess - Initialize a process object
KeAttachProcess - Attach process address space
KeDetachProcess - Detach process address space
KeExcludeProcess - Exclude process from balance set
KeIncludeProcess - Include process in balance set
KeSetPriorityProcess - Set priority of process object

2.2.7.1 Initialize Process

A process object can be initialized with the **KeInitializeProcess** function:

VOID
KeInitializeProcess (
    IN PKPROCESS *Process,*
    IN KPRIORITY *BasePriority,*
    IN KAFFINITY *Affinity,*
    IN ULONG *DirectoryTableBase,*
    IN BOOLEAN *Enable*
    );

Parameters:

Process - A pointer to a control object of type process.

BasePriority - The base priority of the process.

Affinity - The set of processors on which children threads of the process can execute.

DirectoryTableBase - The value that is to be loaded into the Directory Table Base Register when a child thread of the process is dispatched for execution.

Enable - A boolean variable that specifies the default handling mode for data alignment exceptions in children threads.

The process object data structure is initialized with the specified base priority, affinity, directory table base, and default alignment exception handling mode. The process and thread quantum values are initialized with system default values and the process is not considered to be in the balance set.

The *Enable* parameter specifies the default handling mode for data alignment exceptions in children threads. If this parameter is TRUE, then user mode data alignment exceptions are automatically handled by the kernel and are not raised as exceptions. Otherwise, user mode data alignment exceptions are not handled by the kernel and may, or may not, be raised as exceptions depending on host hardware capabilities. Automatic handling of user data alignment exceptions means that the kernel emulates misaligned data references and completes the offending instructions as if no misalignment exception had occurred. Misaligned references in kernel mode are never automatically handled and are always raised as exceptions.

IMPLEMENTATION NOTES:

Certain processors (e.g., the i386) always handle misaligned data in hardware. On these processors, enabling or disabling the automatic handling of data alignment exceptions has no effect. On other processors (e.g., i486, MIPS r3000, r4000SP, and r4000MP) the handling of misaligned data is handled according to the mode established for the respective thread.

2.2.7.2 Attach Process

A thread can attach to another process's address space with the **KeAttachProcess** function:

```
VOID
KeAttachProcess (
    IN PKPROCESS Process
    );
```

<u>Parameters:</u>

Process - A pointer to a control object of type process.

Attaching to another process's address space causes the subject thread to leave the parent process's address space and enter the address space of the target process. This provides the capability for one thread to alter the address space and resources of another process without having complicated data structures or locking protocols.

All of the resources of the target process can be accessed and manipulated by the subject thread while the thread is executing in the target process's address space. This includes the process object table, process private mapping information, working set, etc.

A thread can only attach to one address space at a time. If an attempt is made to attach to a second process's address space while the thread is already attached to another process's address space, then a bug check will occur. In addition, a thread cannot own any mutexes when it attaches to another process's address space. An attempt to do will also cause a bug check to occur.

Attaching to another process's address causes the current APC state of the subject thread to be saved and a new state initialized. While the thread is executing in the attached process's address space, it can receive APCs that were initiated in that address space. APCs that were initiated in the parent process's address space are queued, but not delivered until the thread returns to the parent process's address space.

Attaching to another process's address space does not cause the kernel stack of the subject thread to be locked in memory while the target process is in the balance set. Therefore, both the source and target processes are not allowed to leave the balance set while a thread has the target process's address space attached. This is accomplished by incrementing the process mutex count of both the source and target processes. Artificially incrementing this count prevents each of the processes from being removed from the balance set until their respective counts go to zero. In addition, a thread that has another process's address space attached is allowed to continue execution as if it owned a mutex, if one of the processes is selected for removal from the balance set by the balance set manager.

The execution time of a thread that has another process's address space attached is charged to the target process.

This service will be used by the executive to alter the address map of another process.

2.2.7.3 Detach Process

A thread can detach from another process's address space with the **KeDetachProcess** function:

```
VOID
KeDetachProcess (
        );
```

Detaching from another process's address space causes the subject thread to return to the parent process's address space.

If an attempt is made to detach from another process's address space when the subject thread does not have another process's address space attached, then a bug check will occur. In addition, if a kernel APC is in progress, the kernel APC queue contains an entry, the user APC queue contains an entry, or the thread owns one or more mutexes, then a bug check will also occur.

Detaching from another process's address space causes the saved APC state to be restored and the process mutex counts to be adjusted. If the kernel APC queue is not empty, then an APC_LEVEL software interrupt is requested which will cause the kernel mode APCs to get delivered as appropriate.

2.2.7.4 Exclude Process

A process object can be excluded from the balance set with the **KeExcludeProcess** function:

BOOLEAN
KeExcludeProcess (
    IN PKPROCESS *Process*
    );

<u>Parameters:</u>

Process - A pointer to a control object of type process.

The specified process is excluded from the balance set and its children threads will be removed from further consideration by the thread dispatcher when they no longer own any mutexes and do not have another process's address space attached.

A value of TRUE is returned as the function value, if the process can be immediately removed from the balance set (i.e., none of its children threads own any mutexes or have any address spaces attached). Otherwise, a value of FALSE is returned and the caller must wait on the process's balance set event to determine the exact point when the process can be removed from the balance set.

Processors on which children threads are running (Running state) or about to run (Standby state) are forced to redispatch if their respective threads do not own any mutexes and are not attached to another process's address space.

As ready threads are considered for execution, a test is made to determine if the thread's process is in the balance set. If the thread does not own any mutexes, is not attached to another process' address space, and its parent process is excluded from the balance set, then the thread is removed from its dispatcher ready queue and inserted in the process ready queue. The process ready queue is scanned when the process reenters the balance set and any threads in the process's ready queue are rereadied for execution.

This function is intended for use by the balance set manager.

2.2.7.5 Include Process

A process object can be included in the balance set with the **KeIncludeProcess** function:

VOID
KeIncludeProcess (
    IN PKPROCESS *Process*
    );

Parameters:

Process - A pointer to a control object of type process.

The specified process is included in the balance set and the process's ready queue is scanned. The process ready queue is a list of threads that are ready to run, but which were moved to the process ready queue when they were encountered in one of the dispatcher ready queues. Each thread in the list is removed and readied for execution.

This function is intended for use by the balance set manager.

2.2.7.6 Set Priority Process

The base priority of a process object can be set with the **KeSetPriorityProcess** function:

KPRIORITY
KeSetPriorityProcess (
    IN PKPROCESS *Process*,
    IN KPRIORITY *BasePriority*
    );

Parameters:

Process - A pointer to a control object of type process.

BasePriority - The new base priority of the process object.

The base priority of the specified process is set to the specified value and the priority of all the process's children threads are adjusted as appropriate.

If the new priority is in the realtime class, then the priority of each child thread is set to the new base priority.

If the new priority is in the variable class, then the priority of each thread is computed by taking the current priority of the thread, subtracting out the old base priority, and then adding the new base priority. The computed value is not allowed to cross into the realtime class or go below a priority of 1.

2.2.7.7 Process Accounting Data

As children threads within a process execute, their runtime is accumulated in the parent process object. The units of this summation are clock ticks.

2.2.8 Profile Object

A *profile object* provides the capability to measure the distribution of execution time within a block of code. Both user and system code may be profiled.

Each profile object has three key attributes. First, a profile object applies to an address range or a set of address ranges. The address range is specified by the *RangeBase*, *RangeSize*, and *Process* parameters. *RangeBase* and *RangeSize* select a range of bytes (that is, the area of code) on which to collect profile data. This range is within the address space described by the *Process* parameter. A given profile object either profiles a single address range within a single address space (i.e., applies to one process) or profiles the same single address range across all processes in the system.

Second, a profile object divides the address range being profiled into buckets. Each time a program counter (PC) sample shows the PC to be in one of these buckets, the corresponding counter is incremented. The *BucketSize* parameter controls the size of these buckets.

Third, a profile object reports the number of sampling hits for any given bucket in the corresponding counter. Counters reside in the buffer associated with the profile object when it is created.

Profiling works by sampling the processors PC using a periodic interrupt. The handler for the profiling interrupt searches the list of active profile objects for those with address ranges that match the PC. (I.e., the sampled PC falls within the address range associated with the profile object, and the current process matches the process associated with the profile object.) For each matching profile object, the bucket is computed, and the counter corresponding to the bucket is updated.

When profiling is off, it consumes no processor cycles, and thus may be present in any system. When turned on, the burden it places on the system is inversely proportional to the profiling interval set with **KeSetIntervalProfile** and proportional to the number of active (started) profile objects. A small number of profile objects may be active at any one time.

IMPLEMENTATION NOTE:

On symmetric MP machines, profiling interrupts occur on all processors (at the same rate). On asymmetric machines (i.e., the SystemPro) slave processors do NOT do profiling.

Programming interfaces that support the profile object include:

KeInitializeProfile - Initialize a profile object
KeStartProfile - Start data collection for a profile object
KeStopProfile - Stop data collection for a profile object
KeSetIntervalProfile - Set length of profile interval (globally)
KeQueryIntervalProfile - Query length of profile interval

2.2.8.1 Initialize Profile

A profile object is initialized with **KeInitializeProfile.**

VOID
KeInitializeProfile (
    IN PKPROFILE *Profile*,
    IN PKPROCESS *Process* **OPTIONAL**,
    IN PVOID *RangeBase*,
    IN ULONG *RangeSize*,
    IN ULONG *BucketSize*
    );

Parameters:

Profile - A pointer to a control object of type profile.

Process - If specified, a pointer to a kernel process object that describes the address space to profile. If not specified, then all address spaces are included in the profile.

RangeBase - Address of the first byte of the address range for which profiling information is to be collected.

RangeSize - Size of the address range for which profiling information is to be collected. The *RangeBase* and *RangeSize* parameters are interpreted such that *RangeBase* <= address < *RangeBase*+*RangeSize* generates a profile hit.

BucketSize - Log base 2 of the size of a profiling bucket. Thus, *BucketSize* = 2 yields 4-byte buckets, *BucketSize* = 7 yields 128-byte buckets. All profile hits in a given bucket increment the corresponding counter in *Buffer*. Buckets cannot be smaller than a ULONG.

The profile object is initialized with the specified parameter values, and its state is set to stopped. **KeStartProfile** must be called to actually start profiling.

2.2.8.2 Start Profile

KeStartProfile must be called to start gathering data for a profile object.

BOOLEAN
KeStartProfile (
    IN PKPROFILE *Profile*,
    IN PULONG *Buffer*
    );

Parameters:

Profile - A pointer to a control object of type profile.

Buffer - Array of ULONGs. Each ULONG is a hit counter, which records the number of hits in the corresponding bucket. The *Buffer* must be accessible at DPC_LEVEL and above.

The value TRUE is returned if the profile object is successfully started. FALSE is returned if the object is already in the started state. An exception (STATUS_INSUFFICIENT_RESOURCES) is raised if there are insufficient resources available to make the profile active.

2.2.8.3 Stop Profile

KeStopProfile is called to stop gathering data for a profile object.

BOOLEAN
KeStopProfile (
    IN PKPROFILE *Profile*
    );

Parameters:

Profile - A pointer to a control object of type profile.

TRUE is returned if the profile is successfully stopped, FALSE if it is not already in the started state. Once a profile is stopped, no more updates are written into its buffer.

2.2.8.4 Set System Profile Interval

The time interval between profile interrupts (and thus the profiling rate) is set by calling **KeSetIntervalProfile.**

VOID
KeSetIntervalProfile (
    IN ULONG *Interval*
    );

Parameters:

Interval - The sampling interval in 100ns units.

The actual interval set by the system is the closest available, but may differ significantly. **KeQueryIntervalProfile** returns the actual value in use by the system.

The value is set globally; it affects all profiles on all processors.

IMPLEMENTATION NOTE:

PC-based i386 and i486 machines offer sampling intervals from about 10,000 units (1 millisecond) to 300 units (30 microseconds).

2.2.8.5 Query System Profile Interval

KeQueryIntervalProfile returns the current profile sampling interval.

ULONG
KeQueryIntervalProfile (
    );

The current profile sampling interval is returned in units of 100ns. This is the value the system is actually using, and thus may be different from the value set with **KeSetIntervalProfile**.

3. Wait Operations

Threads synchronize their access to dispatcher objects with object-specific functions and the generic kernel Wait functions. When a thread desires to wait until a dispatcher object attains a Signaled state, it executes one of the kernel Wait functions specifying the dispatcher object as a parameter. If the dispatcher object is not currently in a Signaled state, then the kernel puts the thread in a Waiting state and selects another thread to run on the current processor.

At some future point, a cooperating thread or system operation will cause the specified dispatcher object to attain a state of Signaled. When this occurs, the thread will be given a priority boost and enter the Ready state. The thread will be dispatched for execution according to its priority.

The kernel Wait functions also allow a thread to wait on more than one dispatcher object at a time. The conditions under which the Wait will be satisfied can be specified as *WaitAny* or *WaitAll*.

If *WaitAny* is specified, then the Wait will be satisfied when any of the objects attain a state of Signaled. If *WaitAll* is specified, then the Wait will not be satisfied until all of the objects *concurrently* attain a state of Signaled.

Each Wait operation can optionally specify a timeout value. If a timeout value is specified, then the Wait will be automatically satisfied if the timeout period is exceeded without the Wait being satisfied in the normal manner.

If a timeout value of zero is specified, then no wait will actually occur, but an attempt will be made to satisfy the Wait immediately. If the Wait can be satisfied, then all side effects are performed (e.g. acquiring a mutex). Otherwise, no side effects are performed.

Wait operations can be alertable or nonalertable. If a wait is alertable and the subject thread is alerted while it is waiting, then the wait will be satisifed with a completion status of STATUS_ALERTED.

Wait operations also take a processor mode as a parameter which specifies on whose behalf the Wait is actually occurring. This is required since executive code itself performs the Wait operation and the previous mode of the processor is not necessarily the correct mode. This mode determines what happens when the subject thread is alerted or an APC is queued while the thread is in a Waiting state.

Each Wait operation also takes a Wait reason as a parameter. The Wait reason is an enumerated type supplied by the kernel and is used for debugging system code and for system management functions (i.e., it will be possible to display the reason a thread is in a Waiting state).

Programming interfaces that support wait operations include:

KeWaitForMultipleObjects - Wait for dispatcher objects
KeWaitForSingleObject - Wait for one dispatcher object

3.1 Wait For Multiple Objects

A thread can wait for a set of dispatcher objects with the **KeWaitForMultipleObjects** function:

NTSTATUS
KeWaitForMultipleObjects (
  IN CCHAR *Count*,
  IN PVOID *Objects[]*,
  IN WAIT_TYPE *WaitType*,
  IN KWAIT_REASON *WaitReason*,
  IN KPROCESSOR_MODE *WaitMode*,
  IN BOOLEAN *Alertable*,
  IN PTIME *Timeout* **OPTIONAL**,
  IN PKWAIT_BLOCK *WaitBlockArray* **OPTIONAL**
  );

Parameters:

Count - A count of the number of objects that are to be waited on.

Objects - An array of pointers to dispatcher objects.

WaitType - The type of wait operation that is to be performed (*WaitAny* or *WaitAll*).

WaitReason - The reason for the Wait.

WaitMode - The processor mode on whose behalf the Wait is occurring.

Alertable - A boolean value that specifies whether the Wait is alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which the Wait is to be completed.

WaitBlockArray - An optional pointer to an array of wait blocks that are to be used to describe the wait operation.

Each thread object has a builtin array of wait blocks that can be used to wait on multiple objects concurrently. Whenever possible, the builtin array of wait blocks should be used in a wait multiple operation since no additional wait block storage need be allocated and later deallocated. However, if the number of objects to be waited on concurrently is greater than the number of builtin wait blocks, then the *WaitBlockArray* parameter can be used to specify an alternate set of wait blocks to be used in the wait operation.

If the *WaitBlockArray* parameter is not specified, then the *Count* parameter must be less than or equal to THREAD_WAIT_OBJECTS which defines the number of builtin wait objects. If the *WaitBlockArray* parameter is not specified and the *Count* parameter is greater than THREAD_WAIT_BLOCKS, then a bug check will occur.

If the *WaitBlockArray* parameter is specified, then the *Count* parameter must be less than or equal to MAXIMUM_WAIT_OBJECTS which is the maximum number of objects that can be waited on concurrently. If the *WaitBlockParameter* is specified and the *Count* parameter is greater than MAXIMUM_WAIT_OBJECTS, then a bug check will occur.

The current state for each of the specified objects is examined to determine if the Wait can be satisfied immediately. If the Wait can be satisfied, then necessary side effects are performed on the objects and an appropriate value is returned as the function value. If the Wait cannot be satisfied immediately, and either no timeout value or a nonzero timeout value is specified, then the current thread is put in a Waiting state and a new thread is selected for execution on the current processor.

The *WaitType* parameter specifies the type of wait operation that is to be performed. If the *WaitType* is *WaitAll*, then all of the specified objects must attain a state of Signaled before the Wait will be satisfied. If the *WaitType* is *WaitAny*, then any of the objects must attain a state of Signaled before the Wait will be satisifed.

The reason for the Wait is set to the value specified by the *WaitReason* parameter.

The *WaitMode* parameter specifies on whose behalf the Wait is occurring.

The *Alertable* parameter specifies whether the thread can be alerted while it is in the Waiting state. If the value of this parameter is TRUE and the thread is alerted for a mode that is equal to or more privileged than the Wait mode, then the thread's Wait will be satisfied with a completion status of STATUS_ALERTED.

If the *WaitMode* parameter is *UserMode* and the *Alertable* parameter TRUE, then the thread can also be awakened to deliver a user mode APC. Kernel mode APCs always cause the subject thread to be awakened if the Wait IRQL is zero and there is not a kernel APC in progress.

The *Timeout* parameter is optional. If a timeout value is specified, then the Wait will be automatically satisfied if the timeout occurs before the specified Wait conditions are met.

If a zero timeout value is specified, then the Wait will not actually Wait regardless of whether it can be satisfied or not. An explicit timeout value of zero allows for the testing of a set of Wait conditions, and conditionally performing any side effects if the Wait can be immediately satisifed (e.g. the acquisition of a mutex).

The expiration time of the timeout is expressed as either an absolute time at which the Wait is to be automatically satisifed, or a time that is relative to the current system time. If the value of the *Timeout* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The values returned by the **KeWaitForMultipleObjects** function determine how the Wait was satisfied.

A value in the range of zero to *Count* minus one is returned if the Wait is satisfied by one or more of the dispatcher objects specified by the *Objects* parameter and none of the dispatcher objects satisfying the Wait is an abandoned mutant object. The actual value returned is the index of the object (zero based) in the *Objects* array that satisfied the Wait.

A value in the range of STATUS_ABANDONED to STATUS_ABANDONED plus *Count* minus one is returned if the Wait is satisfied by one or more of the dispatcher objects specified by the *Objects* parameter and one or more of the dispatcher objects satisfying the Wait is an abandoned mutant object. The actual value returned is the index of the object (zero based) in the *Objects* array that satisfied the Wait plus the value of STATUS_ABANDONED.

A value of STATUS_ALERTED is returned if the Wait was completed because the thread was alerted.

If a value of STATUS_TIMEROUT is returned, then timeout occurred before the specified set of wait conditions were met. Note that this value can be returned when an explicit timeout value of zero is specified and the specified set of wait conditions cannot be immediately met.

A value of STATUS_USER_APC is returned if a user mode APC is to be delivered.

3.2 Wait For Single Object

A thread can wait for a single dispatcher object with the **KeWaitForSingleObject** function:

NTSTATUS
KeWaitForSingleObject (
    IN PVOID *Object*,
    IN KWAIT_REASON *WaitReason*,
    IN KPROCESSOR_MODE *WaitMode*,
    IN BOOLEAN *Alertable*,
    IN PTIME *Timeout* **OPTIONAL**
    );

Parameters:

Object - A pointer to a dispatcher object.

WaitReason - The reason for the Wait.

WaitMode - The processor mode on whose behalf the Wait is occurring.

Alertable - A boolean value that specifies whether the Wait is alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which the Wait is to be completed.

The current state of the specified object is examined to determine if the Wait can be satisfied immediately. If the Wait can be satisfied, then necessary side effects are performed on the object and an appropriate value is returned as the function value. If the Wait cannot be satisfied immediately, and either no timeout value or a nonzero timeout value is specified, then the current thread is put in a Waiting state and a new thread is selected for execution on the current processor.

The reason for the Wait is set to the value specified by the *WaitReason* parameter.

The *WaitMode* parameter specifies on whose behalf the Wait is occurring.

The *Alertable* parameter specifies whether the thread can be alerted while it is in the Waiting state. If the value of this parameter is TRUE and the thread is alerted for a mode that is equal to or more privileged than the Wait mode, then the thread's Wait will be satisfied with a completion status of STATUS_ALERTED.

If the *WaitMode* parameter is *UserMode* and the *Alertable* parameter TRUE, then the thread can also be awakened to deliver a user mode APC. Kernel mode APCs always cause the subject thread to be awakened if the Wait IRQL is zero and there is not a kernel APC in progress.

The *Timeout* parameter is optional. If a timeout value is specified, then the Wait will be automatically satisfied if the timeout occurs before the specified Wait conditions are met.

If a zero timeout value is specified, then the Wait will not actually Wait regardless of whether it can be satisfied or not. An explicit timeout value of zero allows for the testing of a set of Wait conditions, and conditionally performing any side effects if the Wait can be immediately satisifed (e.g. the acquisition of a mutex).

The expiration time of the timeout is expressed as either an absolute time at which the Wait is to be automatically satisifed, or a time that is relative to the current system time. If the value of the *Timeout* parameter is negative, then the expiration time is relative. Otherwise, the expiration time is absolute.

The values returned by the **KeWaitForSingleObject** function determine how the Wait was satisfied.

A value of STATUS_SUCCESS is returned if the dispatcher object specified by the *Object* parameter satisfied the Wait.

A value of STATUS_ABANDONED is returned if the dispatcher object specified by the *Object* parameter satisfied the Wait and is a mutant object that was previously abandoned.

A value of STATUS_ALERTED is returned if the Wait was completed because the thread was alerted.

If a value of STATUS_TIMEOUT is returned, then timeout occurred before the specified wait condition was met. Note that this value can be returned when an explicit timeout value of zero is specified and the specified set of wait conditions cannot be immediately met.

A value of STATUS_USER_APC is returned if a user mode APC is to be delivered.

4. Miscellaneous Operations

Several miscellaneous functions are provided to perform hardware-related operations and to provide the operations necessary to debug a multiprocessor operating system.

The exact implementation for some of these operations varies, depending on the particular host architecture. Implementation notes have been provided for these functions.

Programming interfaces that support miscellaneous operations include:

KeBugCheck - Generate bug check halt
KeContextFromKframes - Move machine state to context frames
KeContextToKframes - Move machine state from context frames
KeFillEntryTb - Fill translation buffer entry
KeFlushDcache - Flush data cache
KeFlushEntireTb - Flush entire translation buffer
KeFlushIcache - Flush instruction cache
KeFlushIoBuffers - Flush I/O buffers from the data cache
KeFlushSingleTb - Flush single translation buffer entry
KeFreezeExecution - Freeze processor execution
KeGetCurrentApcEnvironment - Get the current APC environment
KeGetCurrentIrql - Get the current IRQL
KeGetPreviousMode - Get previous processor mode
KeLowerIrql - Lower the current IRQL to the specified value
KeQuerySystemTime - Query the current system time
KeRaiseIrql - Raise the current IRQL to the specified value
KeRundownThread - Run down thread before termination
KeSetSystemTime - Set the current system time
KeStallExecutionProcessor - Stall processor execution
KeUnFreezeExecution - Unfreeze processor execution

4.1 Bug Check

A bug check halt can be generated with the **KeBugCheck** function:

VOID
KeBugCheck (
    IN ULONG *BugCheckCode*
    );

Parameters:

BugCheckCode - A value that specifies the reason for the bug check.

A bug check is a system-detected error that causes a controlled shutdown of the system. The various kernel mode components of the system perform online consistency checking. When an inconsistency is discovered, a bug check is generated.

4.2 Context Frame Manipulation

The kernel trap handler is responsible for saving and restoring machine state when an interrupt, exception, system call, or other trapping condition is detected by system hardware.

Depending on the type of trapping condition, the trap handler may save only the volatile register state, or may save both the volatile and the nonvolatile register state. In addition, the previous processor state and floating point status are also saved.

This state information is saved on the kernel stack in the form of a call frame. Separate call frames are used to store the volatile and the nonvolatile register state. These frames are called the trap frame and exception frame respectively.

A third structure, called a context frame, is constructed from the information contained in the trap and exception frames. This structure contains the complete machine state for a thread of execution.

A context frame is used to specify the initial machine state of a thread, to save the previous machine state when an exception handler is invoked, and to continue the execution of a thread after an exception has been handled.

The kernel supplies two routines to marshal information to/from a context frame.

4.2.1 Move Machine State To Context Frame

Saved machine state can be moved from a trap frame and/or an exception frame to a context frame with the **KeContextFromKframes** function:

```
VOID
KeContextFromKframes (
    IN PKTRAP_FRAME TrapFrame,
    IN PKEXCEPTION_FRAME ExceptionFrame,
    IN OUT PCONTEXT ContextFrame
    );
```

Parameters:

TrapFrame - A pointer to a trap frame.

ExceptionFrame - A pointer to an exception frame.

ContextFrame - A pointer to a context frame.

Saved machine state is moved from the specified trap frame and/or the specified exception frame to the specified context frame. The *ContextFlags* field of the context frame controls the information that is moved.

ContextFlags Field

CONTEXT_CONTROL - Specifies that the processor state information from the trap frame is to be moved to the context frame.

CONTEXT_FLOATING_POINT - Specifies that the floating point register state from the trap and exception frames is to be moved to the context frame.

CONTEXT_INTEGER - Specifies that the integer register state from the trap and exception frames is to be moved to the context frame.

CONTEXT_PIPELINE - Specifies that the floating point pipe state is to be moved from the trap frame to the context frame.

CONTEXT_FULL - Specifies that all of the state information from the trap and exception frames is to be moved to the context frame.

4.2.2 Move Machine State From Context Frame

Saved machine state can be moved from a context frame to a trap frame and/or an exception frame with the **KeContextToKframes** function:

```
VOID
KeContextToKframes (
    IN OUT PKTRAP_FRAME TrapFrame,
    IN OUT PKEXCEPTION_FRAME ExceptionFrame,
    IN PCONTEXT ContextFrame,
    IN ULONG ContextFlags,
    IN KPROCESSOR_MODE PreviousMode
    );
```

Parameters:

TrapFrame - A pointer to a trap frame.

ExceptionFrame - A pointer to an exception frame.

ContextFrame - A pointer to a context frame.

ContextFlags - A set of flags that specifies the state information that is to be moved from the specified context frame to the specified trap frame and/or the specified exception frame.

ContextFlags Flags

CONTEXT_CONTROL - Specifies that the processor state information from the context frame is to be moved to the trap frame.

CONTEXT_FLOATING_POINT - Specifies that the floating point register state from the context frame is to be moved to the trap and exception frames.

CONTEXT_INTEGER - Specifies that the integer register state from the context frame is to be moved to the trap and exception frames.

CONTEXT_PIPELINE - Specifies that the floating point pipe state is to be moved from the context frame to the trap frame.

CONTEXT_FULL - Specifies that all of the state information from the context frame is to be moved to the trap and exception frames.

PreviousMode - The processor mode for which the context frame is specified.

Saved machine state is moved from the specified context frame to the specified trap frame and/or the specified exception frame. The *ContextFlags* parameter specifies the information that is to be moved. The *PreviousMode* parameter determines which bits the caller may specify if the processor state information is being moved to the trap frame.

4.3 Fill Entry Translation Buffer

A page table entry can be inserted into the translation buffer of the current processor with the **KeFillEntryTb** function:

VOID
KeFillEntryTb (
  IN HARDWARE_PTE *Pte[1]*,
  IN PVOID *Virtual*,
  IN BOOLEAN *Invalid*
  );

<u>Parameters:</u>

Pte - A pointer to a page table entry, or a pair of page table entries, that are to be inserted into the translation buffer of the current processor.

Virtual - The virtual address that corresponds to the first page table entry.

Invalid - A boolean value that determines whether a translation buffer entry should be invalidated if the host architecture does not provide a software-managed translation buffer.

This function is intended for use by memory management software for the following cases:

1. A page table entry transitions from the invalid to the valid state.

2. A page table entry transitions from the unmodified (clean) to the modified (dirty) state.

3. A page table entry transitions from the unaccessed to the accessed state.

None of these transitions affects other processors in the configuration; however, they provide the opportunity to optimize the filling of the translation buffer on systems that have a software-managed translation buffer.

If the page table entry is transitioning from the invalid to the valid state, then the *Invalid* parameter should be FALSE. Otherwise, the *Invalid* parameter should be TRUE.

If the specified virtual address is already mapped by the translation buffer, then the contents of the specified page table entry(s) replace the page table entry in the translation buffer. Otherwise, a new translation buffer entry is created that maps the specified virtual address.

IMPLEMENTATION NOTES:

The Intel i860 does not have a software-managed translation buffer. It also cannot invalidate a single translation buffer entry. Therefore, if the *Invalid* parameter is

TRUE, then the entire translation buffer is invalidated. Otherwise, no operation is performed.

The Intel i386 and i486 do not have a software-managed translation buffer. Also neither of these processors can invalidate a single translation buffer entry. Therefore, if the *Invalid* parameter is TRUE, the entire translation buffer is invalidated. Otherwise, no operation is performed.

> \ *The i486 can invalidate a single translation buffer entry, but it is not yet supported.* \

The MIPS r3000, r4000SP, and r4000MP have software-managed translation buffers. Therefore, the specified page table entry either replaces the current translation buffer entry or a new translation buffer entry is created to map the specified virtual address.

4.4 Flush Data Cache

The data cache can be flushed on all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushDcache** function:

VOID
KeFlushDcache (
    IN BOOLEAN *AllProcessors*
    );

Parameters:

> *AllProcessors* - A boolean value that determines which data caches are to be flushed.

This function is intended for use by memory management and device driver software to keep the data cache coherent with DMA I/O operations.

If the *AllProcessors* parameter is TRUE, then the data cache is flushed on all processors in the system. Otherwise, only the data caches on processors running threads that belong to the current thread's process are flushed.

IMPLEMENTATION NOTES:

The Intel i860 employs a writeback data cache with virtual tags that does not maintain coherency with DMA I/O operations. Therefore, this function must flush the data cache.

The Intel i386 and i486 employ data caches that maintain coherency with I/O operations. Therefore, this function performs no operation.

The MIPS r3000 and r4000SP employ data caches that do not maintain coherency with I/O operations. Therefore, the data cache must be flushed for this function.

The MIPS r4000MP employs a data cache that maintains coherency with I/O operations. Therefore, this function performs no operation.

4.5 Flush Entire Translation Buffer

The entire translation buffer can be flushed on all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushEntireTb** function:

VOID
KeFlushEntireTb (
    IN BOOLEAN *Invalid*,
    IN BOOLEAN *AllProcessors*
    );

Parameters:

Invalid - A boolean value that specifies why the translation buffer is being flushed.

AllProcessors - A boolean value that determines which translation buffers are to be flushed.

This function is intended for use by memory management software when virtual pages are deleted, removed from the process working set, or their protection is changed. Normally, the entire translation buffer is not flushed when virtual pages are removed from the process working set. However, when a number of pages are removed all at once, it is more efficient to simply flush the entire translation buffer rather than flush individual entries.

If the value of the *Invalid* parameter is TRUE, then the translation buffer is being flushed because one or more pages have become invalid and not present in memory. If the value of the *Invalid* parameter is FALSE, then the translation buffer is being flushed because the protection on one or more pages has been changed.

If the *AllProcessors* parameter is TRUE, then the entire translation buffer is flushed on all processors in the system. Otherwise, only the translation buffers on processors running threads that belong to the current thread's process are flushed.

IMPLEMENTATION NOTE:

The Intel i860 employs a data cache with virtual tags. It also cannot flush the translation buffer without also flushing the instruction cache. If the *Invalid* parameter is TRUE, then the data cache is flushed in addition to flushing the instruction cache and invalidating the translation buffer. Otherwise, the instruction cache is flushed and the translation buffer is invalidated.

The Intel i386 and i486 flush the translation buffer for this function.

The MIPS r3000, r4000SP, and r4000MP flush the random part of the software-managed translation buffer for this function. The fixed part of the translation buffer is not affected.

4.6 Flush Instruction Cache

The instruction cache can be flushed on all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushIcache** function:

VOID
KeFlushIcache (
    IN BOOLEAN *AllProcessors*
    );

Parameters:

> *AllProcessors* - A boolean value that determines which instruction caches are to be flushed.

This function is intended for use by system debuggers. When a breakpoint is inserted in system code, the instruction caches of all processors in the system must be flushed. If a breakpoint is placed in process code, then only the instruction caches of processors executing threads that belong to current thread's process need to be flushed.

The executive also exports this function for use by code that modifies the instruction stream. After each such modification, and before attempting to execute the modified instructions, the instruction cache must be flushed.

If the *AllProcessors* parameter is TRUE, then the instruction cache is flushed on all processors in the system. Otherwise, only the instruction caches on processors running threads that belong to the current thread's process are flushed.

IMPLEMENTATION NOTES:

The Intel i860 does not maintain coherency between the data and instruction caches. It also cannot flush the instruction cache without invalidating the translation buffer. Therefore, the instruction cache is flushed and the translation buffer is invalidated for this function.

The Intel i386 and i486 maintain coherency between the data and instruction caches. Therefore, no operation is performed for this function.

The MIPS r3000, r4000SP, and r4000MP do not maintain coherency between the instruction and data caches. Therefore, the instruction cache is flushed for this function.

4.7 Flush I/O Buffers

The memory region occupied by an I/O buffer can be flushed from both the instruction and data caches of all processors in the system with the **KeFlushIoBuffers** function:

VOID
KeFlushIoBuffers (
    IN PMDL *Mdl*,
    IN BOOLEAN *ReadOperation*
    );

Parameters:

Mdl - A pointer to a memory descriptor list that describes the areas of memory occupied by the I/O buffer.

ReadOperation - A boolean value that determines whether the flush is being performed for a read operation.

This function is intended for use by device drivers and affects all processors in the system.

If the *ReadOperation* parameter is TRUE, then the I/O operation is reading information into memory that may be valid in the instruction and data caches. If the *ReadOperation* parameter is FALSE, then the I/O operation is writing data from memory to a device and information may be present in the data cache and not in memory.

IMPLEMENTATION NOTES:

The Intel i860 employs a writeback data cache and an instruction cache that do not maintain coherency with I/O operations. Therefore, the data cache must be flushed for both read and write operations. The Intel i860 also cannot flush the instruction cache without invalidating the translation buffer. Therefore, if the *ReadOperation* parameter is TRUE, then the instruction cache is flushed and the translation buffer is also invalidated for this function.

The Intel i386 and i486 maintain data and instruction cache coherency with I/O operations. Therefore, no operation is performed for this function.

> \ *The i486 has a write buffer which may have to be flushed before all I/O operations.* \

The MIPS r3000 employs a write-through data cache and does not maintain coherency with I/O operations for either of the instruction or data caches. Therefore, if the *ReadOperation* parameter is TRUE, then both the instruction and data caches must be flushed. Otherwise, no operation is performed for this function.

> \ *The r3000 has a write buffer which must be flushed before all I/O operations.* \

The MIPS r4000SP employs a writeback data cache and an instruction cache that do not maintain coherency with I/O operations. Therefore, the data cache must be flushed for both read and write operations. In addition, if the *ReadOperation* parameter is TRUE, then the instruction cache is also flushed for this function.

The MIPS r4000MP employs a writeback data cache that maintains coherency with I/O operations. However, cache coherency is not maintained for the instruction cache with I/O operations. Therefore, if the *ReadOperation* parameter is TRUE, then the instruction cache is flushed. Otherwise, no operation is performed for this function.

4.8 Flush Single Translation Buffer Entry

A single entry can be flushed from the translation buffer of all processors, or only that set of processors that are currently executing threads that belong to the current thread's process with the **KeFlushSingleTb** function:

HARDWARE_PTE
KeFlushSingleTb (
    IN PVOID *Virtual*,
    IN BOOLEAN *Invalid*,
    IN BOOLEAN *AllProcessors*,
    IN PHARDWARE_PTE *PtePointer*,
  IN HARDWARE_PTE *PteValue*
  );

Parameters:

Virtual - A virtual address that is within the page whose translation buffer entry is to be flushed.

Invalid - A boolean value that specifies why the translation buffer is being flushed.

AllProcessors - A boolean value that determines which translation buffers are to be flushed.

PtePointer - A Pointer to a page table entry which is to be updated with the new PteValue.

PteValue - The new Pte value.

Return Value:

The contents of the page table entry *PtePointer* refers to before the entry is set to *PteValue*.

This function is intended for use by virtual memory management software when a virtual page is deleted, removed from the process working set, or its protection is changed. If several virtual pages are removed from a process's address space at once or their protection is changed, then it may be more efficient to use the **KeFlushEntireTb** function.

If the value of the *Invalid* parameter is TRUE, then the translation buffer is being flushed because a page has become invalid and is not present in memory. If the value of the *Invalid* parameter is FALSE, then the translation buffer is being flushed because the protection on a page has been changed.

If the *AllProcessors* parameter is TRUE, then the specified translation buffer entry is flushed on all processors in the system. Otherwise, only the specified translation buffer entry on process running threads that belong to the current thread's processor are flushed.

IMPLEMENTATION NOTE:

The Intel i860 employs a data cache with virtual tags. It also cannot invalidate a single entry from the translation buffer nor can it invalidate the translation buffer without also flushing the instruction cache. If the *Invalid* parameter is TRUE, then the data cache is flushed in addition to flushing the instruction cache and invalidating the translation buffer. Otherwise, the instruction cache is flushed and the translation buffer is invalidated.

The Intel i386 and i486 cannot flush a single entry from the translation buffer. Therefore, the entire translation buffer is invalidated for this function.

> \ *The i486 can invalidate a single translation buffer entry, but it is not yet supported.* \

The MIPS r3000, r4000SP, and r4000MP provide the capability to flush a single entry from the random part of the software-managed translation buffer. Therefore, a single translation buffer entry is invalidated for this function.

4.9 Freeze Execution

The execution of all other processors in the system, excluding the current processor, can be frozen with the **KeFreezeExecution** function:

```
KIRQL
KeFreezeExecution (
    );
```

The IRQL is raised to the highest level, the execution of all other processors in the host configuration is frozen, and the previous IRQL is returned as the function value.

This function does not return control to the caller until the execution of all other processors has been frozen. It is intended for use by system debuggers and should be called whenever the debugger is entered so that a consistent picture of the multiprocessor system can be examined and modified.

4.10 Get Current APC Environment

The APC execution environment for the current thread can be obtained with the **KeGetCurrentApcEnvironment** function:

KAPC_ENVIRONMENT
KeGetCurrentApcEnvironment (
    );

The APC execution environment is obtained from the current thread and returned as the function value.

Possible values that can be returned by this function include:

- o OriginalApcEnvironment - The current APC environment is the thread's parent process.
- o AttachedApcEnvironment - The current APC environment is a process that has been attached by the current thread.

4.11 Get Current IRQL

The current IRQL can be obtained with the **KeGetCurrentIrql** function:

KIRQL
KeGetCurrentIrql (
    );

The current IRQL is returned as the function value.

4.12 Get Previous Mode

The previous processor mode can be obtained with the **KeGetPreviousMode** function:

KPROCESSOR_MODE
KeGetPreviousMode (
    );

The previous processor mode is obtained from the processor status. This function can be used to determine the previous processor mode during a system service.

4.13 Lower IRQL

The current IRQL can be lowered with the **KeLowerIrql** function:

VOID
KeLowerIrql (
    IN KIRQL *NewIrql*
    );

<u>Parameters:</u>

NewIrql - The new IRQL value.

If the new IRQL is greater than the current IRQL, then a bug check will occur. Otherwise, the current IRQL is set to the specified value.

4.14 Query System Time

The current system time can be queried with the **KeQuerySystemTime** function:

VOID
KeQuerySystemTime (
    OUT PTIME *CurrentTime*
    );

<u>Parameters:</u>

CurrentTime - A pointer to a variable that receives the current system time.

This function returns the current system time in 100ns units. It is the responsibility of the executive to maintain the correspondence between system time and external time as seen by a user of the system.

4.15 Raise IRQL

The current **IRQL** can be raised with the **KeRaiseIrql** function:

KIRQL
KeRaiseIrql (
    IN KIRQL *NewIrql*
    );

<u>Parameters:</u>

NewIrql - The new IRQL value.

If the new IRQL is less than the current IRQL, then a bug check will occur. Otherwise, the current IRQL is set to the specified value.

4.16 Run Down Thread

Data structures for the current thread that must be guarded by the dispatcher database lock can be run down with the **KeRundownThread** function:

VOID
KeRundownThread (
    );

This function is intended for use just prior to terminating a thread. It run downs appropriate data structures and performs operations necessary to terminate the thread.

Operations performed include:

1. Processing of the mutant ownership list which causes each mutant object owned by the current thread to be released with an abandoned status.

4.17 Set System Time

The current system time can be set with the **KeSetSystemTime** function:

VOID
KeSetSystemTime (
    IN PTIME *NewTime*,
    OUT PTIME *OldTime*,
    );

Parameters:

NewTime - A pointer to a variable that specifies the new system time.

OldTime - A pointer to a variable that receives the previous system time.

This function returns the previous system time in 100ns units and sets the system time to the specified value. It is the responsibility of the executive to maintain the correspondence between system time and external time as seen by a user of the system.

4.18 Stall Execution

The execution of the current processor can be stalled with the **KeStallExecutionProcessor** function:

VOID
KeStallExecutionProcessor (
    IN ULONG *MicroSeconds*
    );

Parameters:

MicroSeconds - The number of microseconds for which execution is to be stalled.

This function stalls the execution of the current processor by executing a processor-dependent routine that busy waits at least the specified number of microseconds, but not significantly longer.

This routine is intended for use by device drivers and other software that must wait a short interval which is less than a clock tick, but larger than a few instructions.

IMPLEMENTATION NOTES:

This function is guaranteed to busy wait for at least the number of specified microseconds and is calibrated at system initialization. Long intervals tend to be very accurate, whereas, short intervals may busy wait for a period that is slightly longer than the specified number of microseconds.

4.19 Unfreeze Execution

The execution of all other processors in a host configuration, excluding the current processor, can be resumed with the **KeUnfreezeExecution** function:

VOID
KeUnfreezeExecution (
    IN KIRQL *Irql*
    );

Parameters:

Irql - The previous IRQL value that is to be restored.

The execution of all processors in the system, excluding the current processor, is unfrozen, the previous IRQL is restored, and the instruction cache of each processor in the configuration is flushed.

This function is intended for use by system debuggers and should be called when execution is to be continued after entering the debugger and calling **KeFreezeExecution** function. Before the execution of an unfrozen processor is continued, its instruction cache and translation buffer are flushed.

5. Intel x86 Specific Functions.

There is a small set of special functions peculiar to the Intel x86 family of processors, which are necessary to fully exploit those processors. These functions are used primarily to manipulate x86 specific control structures, such as the Ldt.

Programming interfaces:

Ke386SetLdtProcess - Set Ldt for a process
Ke386SetDescriptorProcess - Set entry in Ldt a process

5.1 Load an Ldt for a process.

An Ldt (Local Descriptor Table) can be made the active Ldt for a process with Ke386SetLdtProcess:

```
VOID
Ke386SetLdtProcess (
    PKPROCESS Process,
    PLDT_ENTRY Ldt[],
    ULONG      Limit
    );
```

Parameters:

Process - Pointer to KPROCESS object describing the process for which the Ldt is to be set.

Ldt - Pointer to an array of LDT_ENTRYs (that is, a pointer to an Ldt.

Limit - Ldt limit (must be 0 mod 8)

The specified LDT (which may be null) will be made the active Ldt of the specified process, for all threads thereof, on whichever processors they are running. The change will take effect before the call returns.

An Ldt address of NULL or a Limit of 0 will cause the process to receive the NULL Ldt.

This function only exists on i386 and i386 compatible processors.

No checking is done on the validity of Ldt entries.

IMPLEMENATION NOTES:

While a single Ldt structure can be shared among processes, any edits to the Ldt of one of those processes will only be synchronized for that process. Thus, processes other than the one the change is applied to may not see the change correctly.

5.2 Set and Entry in a Process's Ldt.

An individual entry in the Ldt of a process may be edited with Ke386SetDescriptorProcess:

```
VOID
Ke386SetDescriptorProcess (
      PKPROCESS Process,
      ULONG         Offset,
      LDT_ENTRY  LdtEntry
      );
```

Parameters:

Process - Pointer to KPROCESS object describing the process for which the descriptor edit is to be performed.

Offset - Byte offset into the Ldt of the descriptor to edit. Must be 0 mod 8.

LdtEntry - Value to edit into the descriptor in hardware format. No checking is done on the validity of this item.

The specified LdtEntry (which could be 0, not present, etc) will be edited into the specified Offset in the Ldt of the specified Process. This will be synchronized across all the processors executing the process. The edit will take affect on all processors before the call returns.

Get an Entry from a Thread's Gdt.

Get an Entry from a Thread's Gdt.

5.3 Get an Entry from a Thread's Gdt.

An individual entry in the Gdt of a thread may be obtained using Ke386GetGdtEntryThread:

VOID
Ke386GetGdtEntryThread (
  IN PKTHREAD *Thread*,
  IN ULONG *Offset*,
  IN PGDT_ENTRY *Descriptor*
  );

Parameters:

Thread — Supplies a pointer to the thread from whose Gdt the entry is to come.

Offset — Supplies the descriptor number of the descriptor to return. This value must be 0 mod 8.

Descriptor — Returns the descriptor contents

The Gdt entry specified by *Selector* will be copied from the specified thread's Gdt, into *Descriptor*.

For descriptors that don't exist when the thread is not running (KGDT_R3_TEB, and KGDT_LDT), the descriptor values will be "materialized".

For descriptors that are processor specific, rather than thread specific, the current processor's value will be returned.

For all other Gdt descriptors, the descriptor will be copied from the Gdt.

Revision History:

Original Draft 1.0, March 8, 1989

Revision 1.1, March 16, 1989

1. Add text to describe the muxwait object.
2. Add text to describe the interrupt object.
3. Add text to describe the power notify object.
4. Add text to describe the power status object.
5. Add text to describe the generic wait functions.
6. Addition of text to describe the miscellaneous functions.
7. Add text to overview of document.

Revision 1.2, March 29, 1989

1. Change **KeDelayExecution** to return a wait completion value.
2. Complete section on multiprocessor synchronization.
3. Complete section on device queue object.
4. Delete muxwait object and replace with a wait multiple function that takes an array of pointer to dispatcher objects as a parameter.

Revision 1.3, April 18,1989

1. Alphabetically order section on miscellaneous functions.
2. Add **KeBugCheck**, **KeLowerIrql**, and **KeRaiseIrql** miscellaneous functions.
3. A thread will start execution at IRQL APC_LEVEL rather than with APCs disabled.
4. Returning from the executive thread start up routine will cause a thread to enter user mode provide that a user mode context was supplied when the thread was initialized.

5. Add three parameters to thread initialization to optionally describe user mode context.

6. Delete builtin user mode alert APC. Alerting a thread that is waiting alertable causes a wait completion status of ALERTED to be returned.

7. Replace all reference to DPC_LEVEL with DISPATCH_LEVEL.

8. Put interrupt level names in hardware interrupt table.

9. Add pointers to the PRCB which point to the time expiration and power notify DPC's that are system wide.

10. Change thread context to include ten pipeline state registers rather than six.

11. Change **KeResumeThread** and **KeSuspendThread** to return a CHAR rather than a UCHAR.

12. Delete voluntary and preemption wait counters from thread object.

13. Reduce number of APC and DPC system parameters to two.

14. If a device is Not-Busy, then release device queue spin lock but do not lower IRQL before returning.

15. Allow interrupt service routine to user the floating point registers for block moves and graphics functions.

16. If a thread is awakened to deliver a user mode APC, then return a status of USER_APC.

Revision 1.4, May 4, 1989

1. Delete increment parameter from **KeReleaseMutex**.

2. Change *Count* and *Limit* parameters of **KeInitializeSemaphore** from ULONG to LONG.

3. Change **KeReadStateSemaphore** to return a LONG rather than a ULONG.

4. Change **KeReleaseSemaphore** to return a LONG rather than a ULONG and change the type of the *Adjustment* parameters from ULONG to LONG.

5. Set the value of a semaphore to the maximum value if an attempt is made to adjust the count of a semaphore above the limit.

6. Add system startup routine to **KeInitializeThread** for executive level initialization.

7. Change the wait functions to return NTSTATUS rather than ULONG.

8. Change the type of the *WaitType* parameter from KWAIT_TYPE to WAIT_TYPE.

Revision 1.5, May 8, 1989

1. Change data type of the **KeBugCheck** parameter to ULONG.

2. Add *Invalid* parameter to **KeFlushEntireTb** and **KeFlushSingleTb** to allow specification of why the flush is being performed.

Revision 1.6, August 14, 1989

1. General correction of typos and grammatical errors as suggested by Helenc review. Clarification and rewrite of several sections.

2. Reorganization of section 1.0 with the deletion of redundant information.

3. Added miscellaneous functions to get the previous processor mode, get the current IRQL, and to set the current IRQL.

4. The interrupt object section was extensively rewritten to match the actual implementation.

5. The time parameter in the **KeDelayExecution** function was changed to a pointer to a time value.

Revision 1.7, November 15, 1989

1. Delete **KeSetCurrentIrql** function which was an optimization of the **KeRaiseIrql** function that didn't require saving the old **IRQL**.

2. Add functions **KeContextToKframes** and **KeContextFromKframes** to move information in the trap frame and the exception frame to/from the context frame.

3. Add priority increment parameter to the **KeInsertQueueApc** function.

4. Change **KeInitializeThread** function to replace the optional trap and exception frame parameters with an optional context frame argument. If the context frame parameter is specified, then it is assumed that the thread will execute in user mode.

5. Change **KeAcquireSpinlock** function to return the old IRQL as an output parameter and delete the Wait parameter from the **KeReleaseSpinlock** function.

6. Change **KeInsertDeviceQueue** and **KeInsertByKeyDevice** functions to neither raise nor lower IRQL.

7. Split the event object into two types of event objects: synchronization and notification.

8. Change the definition of the state of an event object to be a count.

9. Add a parameter to the **KeInitializeApc** function to specify the APC execution environment. Add a function (**KeGetApcEnvironment**) that returns the current APC environment.

Revision 1.8, November 16, 1989

1. Add optional parameter to **KeWaitForMultipleObjects** that allows more than the builtin number of objects to be waited on concurrently.

2. Add abandoned return status from **KeWaitForMultipleObjects** when one or more of the dispatcher objects satisfying the Wait is an abandoned mutant object.

3. Add new mutant object which provides for user level mutexes. Add the functions **KeInitializeMutant**, **KeReadStateMutant**, and **KeReleaseMutant** to manipulate the mutant object.

Revision 1.9, November 17, 1989

1. Add **KeRundownThread** function to provide kernel thread rundown when a thread is deleted.

2. Minor edits and corrections.

Revision 1.10, January 6, 1990

1. Change name of **KeReadSystemTime** to **KeQuerySystemTime**.

2. Add miscellaneous kernel function (**KeSetSystemTime**) to set the system time.

Revision 1.11, June 6, 1990

1. Change text that explains how a binary semaphore can be used like a synchronization event to include a statement that the semaphore cannot be over Signaled.

2. Change the definition of **KeInitializeSemaphore** to omit any checks on the limit and initial value of the semaphore.

3. Change the semantics of release semaphore such that an attempt to over Signal the semaphore does not cause the current count to be set to the maximum value. The current count remains unchanged and the exception STATUS_SEMAPHORE_COUNT_EXCEEDED is raised.

4. Once set, the abandoned status of a mutant object cannot be cleared and will continually be returned by the kernel wait services. The mutant object, however, will continue to function and ownership can be requested and released.

5. If an attempt is made to release a mutant object by a nonowner with the Abandoned parameter FALSE, then either the exception STATUS_ABANDONED or STATUS_MUTANT_NOT_OWNED will be raised.

6. Remove the hard assignment of Interrupt Request Levels (IRQL's) to kernel functions and replace with symbolic assignments. Explain that IRQL's are hierarchically ordered by priority.

7. Change the return type of **KeResumeThread** and **KeSuspendThread** to ULONG.

8. An attempt to suspend a thread more than MAXIMUM_SUSPEND_COUNT times causes the exception STATUS_SUSPEND_COUNT_EXCEEDED to be raised.

9. Add **KeFreezeThread** and **KeUnfreezeThread** functions to suspend and resume a thread on behalf of the system. These functions are identical to suspend and resume, but are not exported to user mode.

10. Add **KeRundownThread** function to run down appropriate kernel data structures before thread termination.

11. Add **KeStallExecution** function to enable executive software to stall execution for short periods of time.

12. Add **KeFlushIoBuffers** function to flush the memory region occupied by an I/O buffer from the data and instruction caches.

13. Remove text that described the **KeFlushIcache** and **KeFlushDcache** functions as intended for use in systems in which DMA I/O operations do not invalidate caches.

14. Add **KeFillEntryTb** function to update TB entries in systems with software-managed translation buffers.

15. Make explanation of thread context more general and not specific to the Intel i860.

Revision 1.12, September 19, 1990 (Bryan Willman)

1. Add Profile object section.

Revision 1.13, March 11, 1991

1. Change the operation of the power notify object so that a DPC object is not required and make the operation repeatable.

2. Add parameter to **KeInitializeProcess** to specific the default data alignment handling mode for children threads.

3. Add **KeSetAutoAlignmentThread** and **KeQueryAutoAlignmentThread** functions to set and query the data alignment handling mode of the current thread.

4. Change the name of the **KeDelayExecution** function to **KeDelayExecutionThread** and move the explanatory text to the section on thread objects.

5. Change the name of the **KeStallExecution** function to **KeStallExecutionProcessor**.

6. Change REQUEST_LEVEL with IPI_LEVEL.

7. Add **KeQueryBasePriorityThread** and **KeSetBasePriorityThread** functions to set and query the base priority of a thread object.

Revision 1.14, May 2, 1991

1. Add x86 specific section, Ke386SetLdtProcess and Ke386SetDescriptorProcess.

Revision 1.15, May 28, 1991 (daveh)

1. Added Ke386GetThreadGdt

Revision 1.16, June 18, 1991 (bryanwi)

1. Added ShareVector parameter to KeInitializeInterrupt.
2. Applied spelling checker to document.

Revision 1.17, August 7, 1991 (shielint)

1. Made KeFlushSingleTb spec match reality

Revision 1.18, August 8, 1991 (bryanwi)

1. Removed coordinator, added SynchronizeIrql to KeInitializeInterrupt.