Microsoft

Windows® Internals

Part 1

6

SIXTH EDITION

Mark Russinovich
David A. Solomon
Alex Ionescu

Contents at a Glance

Windows Internals, Sixth Edition, Part 1

Windows Internals, Sixth Edition, Part 2 *(available separately)*

CHAPTER 8 I/O System
CHAPTER 9 Storage Management
CHAPTER 10 Memory Management
CHAPTER 11 Cache Manager
CHAPTER 12 File Systems
CHAPTER 13 Startup and Shutdown
CHAPTER 14 Crash Dump Analysis

Chapter 5 Processes, Threads, and Jobs 359

Thread Scheduling

This section describes the Windows scheduling policies and algorithms. The first subsection provides a condensed description of how scheduling works on Windows and a definition of key terms. Then Windows priority levels are described from both the Windows API and the Windows kernel points of view. After a review of the relevant Windows utilities and tools that relate to scheduling, the detailed data structures and algorithms that make up the Windows scheduling system are presented, including a description of common scheduling scenarios and how thread selection, as well as processor selection, occurs.

Overview of Windows Scheduling

Windows implements a priority-driven, preemptive scheduling system—at least one of the highest-priority runnable (ready) threads always runs, with the caveat that certain high-priority threads ready to run might be limited by the processors on which they might be allowed or preferred to run on, a phenomenon called *processor affinity*. Processor affinity is defined based on a given processor group, which collects up to 64 processors. By default, threads can run only on any available processors within the processor group associated with the process (to maintain compatibility with older versions of Windows which supported only 64 processors), but developers can alter processor affinity by using the appropriate APIs or by setting an affinity mask in the image header, while users can use tools to change affinity at runtime or at process creation. However, although multiple threads in a process can be associated with different groups, a thread on its own can run only on the processors available within its assigned group. Additionally, developers can choose to create group-aware applications, which use extended scheduling APIs to associate logical processors on different groups with the affinity of their threads. Doing so converts the process into a multigroup process that can theoretically run its threads on any available processor within the machine.

EXPERIMENT: Viewing Ready Threads

You can view the list of ready threads with the kernel debugger *!ready* command. This command displays the thread or list of threads that are ready to run at each priority level. In the following example, generated on a 32-bit machine with a dual-core processor, two threads are ready to run at priority 8 on the first logical processor, and one thread at priority 10, two threads at priority 9, and three threads at priority 8 are ready to run on the second logical processor. Determining which of these threads get to run on their respective processor is a simple matter of picking the first thread on top of the highest priority queue (thread 857d9030 for logical processor 0, and thread 857c0030 for logical processor 1), but why the queues contain the threads they do is a complex result at the end of several algorithms that the scheduler uses. We will cover this topic later in this section.

```
kd> !ready
Processor 0: Ready Threads at priority 8
    THREAD 857d9030  Cid 0ec8.0e30  Teb: 7ffdd000 Win32Thread: 00000000 READY
    THREAD 855c8300  Cid 0ec8.0eb0  Teb: 7ff9c000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 10
    THREAD 857c0030  Cid 04c8.0378  Teb: 7ffdf000 Win32Thread: fef7f8c0 READY
Processor 1: Ready Threads at priority 9
    THREAD 87fc86f0  Cid 0ec8.04c0  Teb: 7ffd3000 Win32Thread: 00000000 READY
    THREAD 88696700  Cid 0ec8.0ce8  Teb: 7ffa0000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 8
    THREAD 856e5520  Cid 0ec8.0228  Teb: 7ff98000 Win32Thread: 00000000 READY
    THREAD 85609d78  Cid 0ec8.09b0  Teb: 7ffd9000 Win32Thread: 00000000 READY
    THREAD 85fdeb78  Cid 0ec8.0218  Teb: 7ff72000 Win32Thread: 00000000 READY
```

After a thread is selected to run, it runs for an amount of time called a quantum. A quantum is the length of time a thread is allowed to run before another thread at the same priority level is given a turn to run. Quantum values can vary from system to system and process to process for any of three reasons:

- System configuration settings (long or short quantums, variable or fixed quantums, and priority separation)
- Foreground or background status of the process
- Use of the job object to alter the quantum

These details are explained in more details in the "Quantum" section later in the chapter, as well as in the "Job Objects" section).

A thread might not get to complete its quantum, however, because Windows implements a preemptive scheduler: if another thread with a higher priority becomes ready to run, the currently running thread might be preempted before finishing its time slice. In fact, a thread can be selected to run next and be preempted before even beginning its quantum!

The Windows scheduling code is implemented in the kernel. There's no single "scheduler" module or routine, however—the code is spread throughout the kernel in which scheduling-related events occur. The routines that perform these duties are collectively called the kernel's dispatcher. The following events might require thread dispatching:

- A thread becomes ready to execute—for example, a thread has been newly created or has just been released from the wait state.
- A thread leaves the running state because its time quantum ends, it terminates, it yields execution, or it enters a wait state.
- A thread's priority changes, either because of a system service call or because Windows itself changes the priority value.
- A thread's processor affinity changes so that it will no longer run on the processor on which it was running.

At each of these junctions, Windows must determine which thread should run next on the logical processor that was running the thread, if applicable, or on which logical processor the thread should now run on. After a logical processor has selected a new thread to run, it eventually performs a context switch to it. A context switch is the procedure of saving the volatile processor state associated with a running thread, loading another thread's volatile state, and starting the new thread's execution.

As already noted, Windows schedules at the thread granularity. This approach makes sense when you consider that processes don't run but only provide resources and a context in which their threads run. Because scheduling decisions are made strictly on a thread basis, no consideration is given to what process the thread belongs to. For example, if process A has 10 runnable threads, process B has 2 runnable threads, and all 12 threads are at the same priority, each thread would theoretically receive one-twelfth of the CPU time—Windows wouldn't give 50 percent of the CPU to process A and 50 percent to process B.

Priority Levels

To understand the thread-scheduling algorithms, one must first understand the priority levels that Windows uses. As illustrated in Figure 5-14, internally Windows uses 32 priority levels, ranging from 0 through 31. These values divide up as follows:

- Sixteen real-time levels (16 through 31)
- Sixteen variable levels (0 through 15), out of which level 0 is reserved for the zero page thread

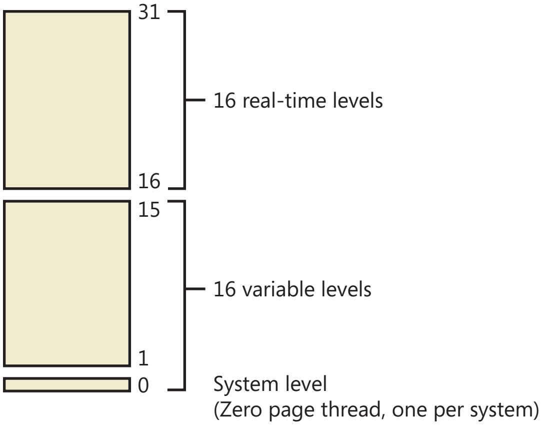


FIGURE 5-14 Thread priority levels

Thread priority levels are assigned from two different perspectives: those of the Windows API and those of the Windows kernel. The Windows API first organizes processes by the priority class to which they are assigned at creation (the numbers represent the internal PROCESS_PRIORITY_CLASS_ index recognized by the kernel): Real-time (4), High (3), Above Normal (7), Normal (2), Below Normal (5), and Idle (1).

It then assigns a relative priority of the individual threads within those processes. Here, the numbers represent a priority delta that is applied to the process base priority: Time-critical (15), Highest (2), Above-normal (1), Normal (0), Below-normal (–1), Lowest (–2), and Idle (–15).

Therefore, in the Windows API, each thread has a base priority that is a function of its process priority class and its relative thread priority. In the kernel, the process priority class is converted to a base priority by using the *PspPriorityTable* and the PROCESS_PRIORITY_CLASS indices shown earlier, which sets priorities of 4, 8, 13, 24, 6, and 10, respectively. (This is a fixed mapping that cannot be changed.) The relative thread priority is then applied as a differential to this base priority. For example, a "Highest" thread will receive a thread base priority of two levels higher than the base priority of its process.

This mapping from Windows priority to internal Windows numeric priority is shown in Table 5-3.

TABLE 5-3 Mapping of Windows Kernel Priorities to the Windows API

Priority Class Relative Priority	Realtime	High	Above Normal	Normal	Below Normal	Idle
Time Critical (+ SATURATION)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (- SATURATION)	16	1	1	1	1	1

You'll note that the Time-Critical and Idle relative thread priorities maintain their respective values regardless of the process priority class (unless it is Realtime). This is because the Windows API requests saturation of the priority from the kernel, by actually passing in 16 or -16 as the requested relative priority (instead of 15 or -15). This is then recognized by the kernel as a request for saturation, and the Saturation field in KTHREAD is set. This causes, for positive saturation, the thread to receive the highest possible priority within its priority class (dynamic or real-time), or for negative saturation, the lowest possible one. Additionally, future requests to change the base priority of the process will no longer affect the base priority of these threads, because saturated threads are skipped in the processing code.

Whereas a process has only a single base priority value, each thread has two priority values: current and base. Scheduling decisions are made based on the current priority. As explained in the following section on priority boosting, the system under certain circumstances increases the priority of threads in the dynamic range (0 through 15) for brief periods. Windows never adjusts the priority of threads in the real-time range (16 through 31), so they always have the same base and current priority.

A thread's initial base priority is inherited from the process base priority. A process, by default, inherits its base priority from the process that created it. This behavior can be overridden on the *CreateProcess* function or by using the command-line start command. A process priority can also be changed after being created by using the *SetPriorityClass* function or various tools that expose that function, such as Task Manager and Process Explorer (by right-clicking on the process and choosing a new priority class). For example, you can lower the priority of a CPU-intensive process so that it does not interfere with normal system activities. Changing the priority of a process changes the thread priorities up or down, but their relative settings remain the same.

Normally, user applications and services start with a normal base priority, so their initial thread typically executes at priority level 8. However, some Windows system processes (such as the session manager, service control manager, and local security authentication process) have a base process priority slightly higher than the default for the Normal class (8). This higher default value ensures that the threads in these processes will all start at a higher priority than the default value of 8.

Real-Time Priorities

You can raise or lower thread priorities within the dynamic range in any application; however, you must have the increase scheduling priority privilege to enter the real-time range. Be aware that many important Windows kernel-mode system threads run in the real-time priority range, so if threads spend excessive time running in this range, they might block critical system functions (such as in the memory manager, cache manager, or other device drivers).

Using the standard Windows APIs, once a process has entered the real-time range, all of its threads (even Idle ones) must run at one of the real-time priority levels. It is thus impossible to mix real-time and dynamic threads within the same process through standard interfaces. This is because the *SetThreadPriority* API calls the native *NtSetInformationThread* API with the *ThreadBasePriority* information class, which allows priorities to remain only in the same range. Furthermore, this information class allows priority changes only in the recognized Windows API deltas of –2 to 2 (or real-time/idle), unless the request comes from CSRSS or a real-time process. In other words, this means that a real-time process does have the ability to pick thread priorities anywhere between 16 and 31, even though the standard Windows API relative thread priorities would seem to limit its choices based on the table that was shown earlier.

However, by calling this API with the *ThreadActualBasePriority* information class, the kernel base priority for the thread can be directly set, including in the dynamic range for a real-time process.

Note As illustrated in Figure 5-15, which shows the interrupt request levels (IRQLs), although Windows has a set of priorities called real-time, they are not real-time in the common definition of the term. This is because Windows doesn't provide true, real-time operating system facilities, such as guaranteed interrupt latency or a way for threads to obtain a guaranteed execution time.

Interrupt Levels vs. Priority Levels

As illustrated in Figure 5-15 of the interrupt request levels (IRQLs) for a 32-bit system, threads normally run at IRQL 0 (called *passive level*, because no interrupts are in process and none are blocked) or IRQL 1 (APC level). (For a description of how Windows uses interrupt levels, see Chapter 3.) User-mode code always runs at passive level. Because of this, no user-mode thread, regardless of its priority, can ever block hardware interrupts (although high-priority, real-time threads can block the execution of important system threads).

Threads running in kernel mode, although initially scheduled at passive level or APC level, can raise IRQL to higher levels—for example, while executing a system call that involves thread dispatching, memory management, or input/output. If a thread does raise IRQL to dispatch level or above, no further thread-scheduling behavior will occur on its processor until it lowers IRQL below dispatch level. A thread executing at dispatch level or above blocks the activity of the thread scheduler and prevents thread context switches on its processor.

A thread running in kernel mode can be running at APC level if it is running a special kernel APC; or it can temporarily raise IRQL to APC level to block the delivery of special kernel APCs. (For more information on APCs, see Chapter 3.) However, executing at APC level does not alter the scheduling behavior of the thread vs. other threads; it affects only the delivery of kernel APCs to that thread. In fact, a thread executing in kernel mode at APC level can be preempted in favor of a higher priority thread running in user mode at passive level.

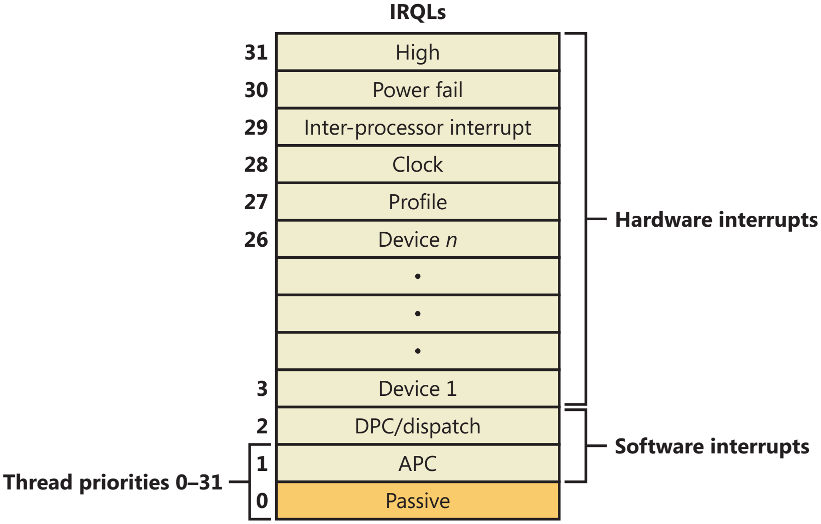


FIGURE 5-15 Thread priorities vs. IRQLs on an x86 system

Using Tools to Interact with Priority

You can change (and view) the base-process priority with Task Manager and Process Explorer. You can kill individual threads in a process with Process Explorer (which should be done, of course, with extreme care).

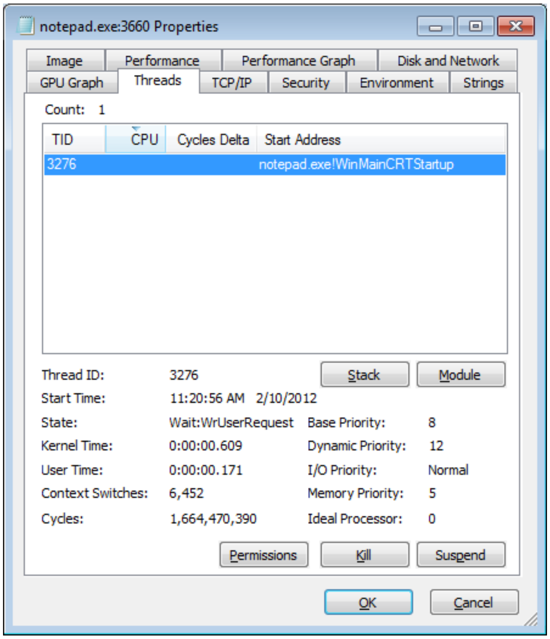
You can view individual thread priorities with the Performance Monitor, Process Explorer, or WinDbg. Although it might be useful to increase or lower the priority of a process, it typically does not make sense to adjust individual thread priorities within a process, because only a person who thoroughly understands the program (in other words, typically only the developer himself) would understand the relative importance of the threads within the process.

The only way to specify a starting priority class for a process is with the start command in the Windows command prompt. If you want to have a program start every time with a specific priority, you can define a shortcut to use the start command by beginning the command with **cmd /c**. This runs the command prompt, executes the command on the command line, and terminates the command prompt. For example, to run Notepad in the low-process priority, the shortcut is **cmd /c start /low Notepad.exe**.

EXPERIMENT: Examining and Specifying Process and Thread Priorities

Try the following experiment:

1. From an elevated command prompt, type **start /realtime notepad**. Notepad should open.
2. Run Process Explorer, and select Notepad.exe from the list of processes. Double-click on Notepad.exe to show the process properties window, and then click on the Threads tab, as shown here. Notice that the dynamic priority of the thread in Notepad is *24*. This matches the real-time value shown in the following image.



Count: 1

TID	CPU	Cycles Delta	Start Address
3276			notepad.exe!WinMainCRTStartup

Thread ID: 3276

Start Time: 11:20:56 AM 2/10/2012

State: Wait:WrUserRequest

Kernel Time: 0:00:00.609

User Time: 0:00:00.171

Context Switches: 6,452

Cycles: 1,664,470,390

Base Priority: 8

Dynamic Priority: 12

I/O Priority: Normal

Memory Priority: 5

Ideal Processor: 0

3. Task Manager can show you similar information. Press Ctrl+Shift+Esc to start Task Manager, and click on the Processes tab. Right-click on the Notepad.exe process, and select the Set Priority option. You can see that Notepad's process priority class is Realtime, as shown in the following dialog box:

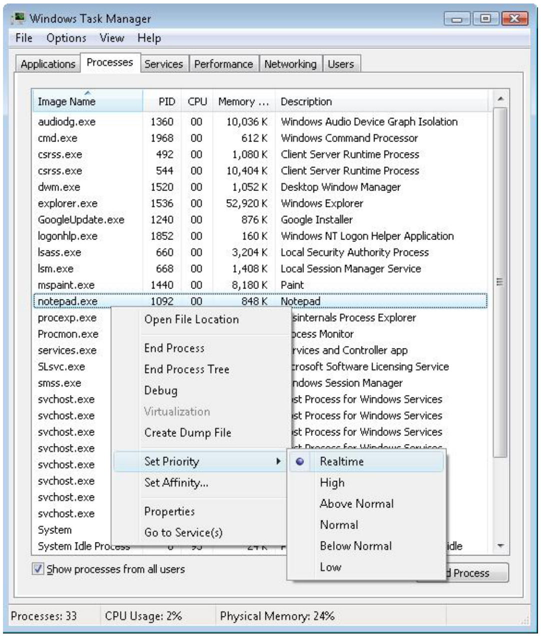


Image Name	PID	CPU	Memory ...	Description
audiodg.exe	1360	00	10,036 K	Windows Audio Device Graph Isolation
cmd.exe	1968	00	612 K	Windows Command Processor
csrss.exe	492	00	1,080 K	Client Server Runtime Process
csrss.exe	544	00	10,404 K	Client Server Runtime Process
dwm.exe	1520	00	1,052 K	Desktop Window Manager
explorer.exe	1536	00	52,920 K	Windows Explorer
GoogleUpdate.exe	1240	00	876 K	Google Installer
logonhlp.exe	1852	00	160 K	Windows NT Logon Helper Application
lsass.exe	660	00	3,204 K	Local Security Authority Process
lsm.exe	668	00	1,408 K	Local Session Manager Service
mspaint.exe	1440	00	8,180 K	Paint
notepad.exe	1092	00	848 K	Notepad
procexp.exe				sinternals Process Explorer
Procmon.exe				ocess Monitor
services.exe				rvices and Controller app
SLsvc.exe				rosoft Software Licensing Service
smss.exe				ndows Session Manager
svchost.exe				st Process for Windows Services
svchost.exe				st Process for Windows Services
svchost.exe				st Process for Windows Services
svchost.exe				
svchost.exe				
svchost.exe				
svchost.exe				
svchost.exe				
System				
System Idle Process				idle

☑ Show processes from all users

Processes: 33 CPU Usage: 2% Physical Memory: 24%

Windows System Resource Manager

Windows Server 2008 R2 Standard Edition and higher SKUs include an optionally installable component called Windows System Resource Manager (WSRM). It permits the administrator to configure policies that specify CPU utilization, affinity settings, and memory limits (both physical and virtual) for processes. In addition, WSRM can generate resource utilization reports that can be used for accounting and verification of service-level agreements with users.

Policies can be applied for specific applications (by matching the name of the image with or without specific command-line arguments), users, or groups. The policies can be scheduled to take effect at certain periods or can be enabled all the time.

After you set a resource-allocation policy to manage specific processes, the WSRM service monitors CPU consumption of managed processes and adjusts process base priorities when those processes do not meet their target CPU allocations.

The physical memory limitation uses the function *SetProcessWorkingSetSizeEx* to set a hard-working set maximum. The virtual memory limit is implemented by the service checking the private virtual memory consumed by the processes. (See Chapter 10 in Part 2 for an explanation of these memory limits.) If this limit is exceeded, WSRM can be configured to either kill the processes or write an entry to the Event Log. This behavior can be used to detect a process with a memory leak before it consumes all the available committed memory on the system. Note that WSRM memory limits do not apply to Address Windowing Extensions (AWE) memory, large page memory, or kernel memory (nonpaged or paged pool).

Thread States

Before you can comprehend the thread-scheduling algorithms, you need to understand the various execution states that a thread can be in. The thread states are as follows:

- **Ready** A thread in the ready state is waiting to execute (or ready to be in-swapped after completing a wait). When looking for a thread to execute, the dispatcher considers only the pool of threads in the ready state.
- **Deferred ready** This state is used for threads that have been selected to run on a specific processor but have not actually started running there. This state exists so that the kernel can minimize the amount of time the per-processor lock on the scheduling database is held.
- **Standby** A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exist, the dispatcher performs a context switch to this thread. Only one thread can be in the standby state for each processor on the system. Note that a thread can be preempted out of the standby state before it ever executes (if, for example, a higher priority thread becomes runnable before the standby thread begins execution).

- **Running** Once the dispatcher performs a context switch to a thread, the thread enters the running state and executes. The thread's execution continues until its quantum ends (and another thread at the same priority is ready to run), it is preempted by a higher priority thread, it terminates, it yields execution, or it voluntarily enters the waiting state.
- **Waiting** A thread can enter the waiting state in several ways: a thread can voluntarily wait for an object to synchronize its execution, the operating system can wait on the thread's behalf (such as to resolve a paging I/O), or an environment subsystem can direct the thread to suspend itself. When the thread's wait ends, depending on the priority, the thread either begins running immediately or is moved back to the ready state.
- **Transition** A thread enters the transition state if it is ready for execution but its kernel stack is paged out of memory. Once its kernel stack is brought back into memory, the thread enters the ready state.
- **Terminated** When a thread finishes executing, it enters the terminated state. Once the thread is terminated, the executive thread object (the data structure in a nonpaged pool that describes the thread) might or might not be deallocated. (The object manager sets the policy regarding when to delete the object.)
- **Initialized** This state is used internally while a thread is being created.

Table 5-4 describes the state transitions for threads, and Figure 5-16 illustrates a simplified version. (The numeric values shown represent the value of the thread-state performance counter.) In the simplified version, the Ready, Standby, and Deferred Ready states are represented as one. This reflects the fact that the Standby and Deferred Ready states act as temporary placeholders for the scheduling routines. These states are almost always very short-lived; threads in these states always transition quickly to Ready, Running, or Waiting. More details on what happens at each transition are included later in this section.

TABLE 5-4 Thread States and Transitions

	Init	Ready	Running	Standby	Terminated	Waiting	Transition	Deferred Ready	
Init									A thread becomes Initialized during the first few moments of its creation (*KeStartThread*).
Ready								A thread is added in the dispatcher-ready database of its ideal processor.	
Running		Selected by *KiSearchForNewThread*		Picked up for execution by local CPU		Preemption after wait satisfaction			

	Init	**Ready**	**Running**	**Standby**	**Terminated**	**Waiting**	**Transition**	**Deferred Ready**	
Standby		Selected by *KiSelectNextThread*						Selected *by KiDeferredReadyThread* for remote CPU	
Terminated	Killed before *PspInsertThread* finished		Killed						A thread can kill only itself. It must be in the Running state before entering KeTerminateThread.
Waiting			Thread enters a wait						Only running threads can wait.
Transition						Kernel stack no longer resident			Only waiting threads can transition.
Deferred Ready	Last step in *PspInsertThread*	Affinity change	Thread becomes preempted (if old processor is no longer available)	Affinity change		Wait satisfaction (but no preemption)	Kernel stack swap-in completed		

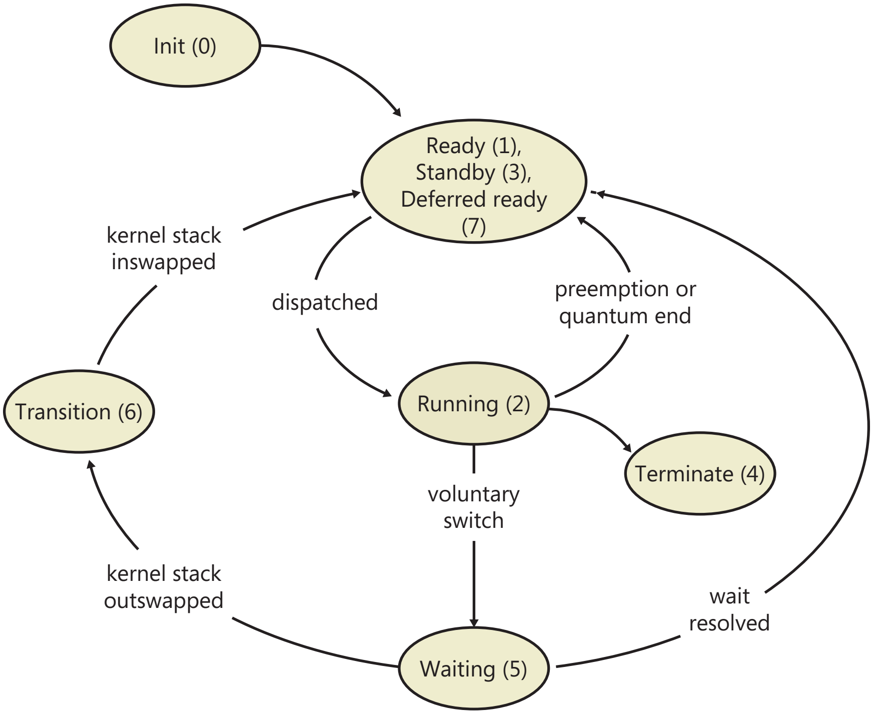
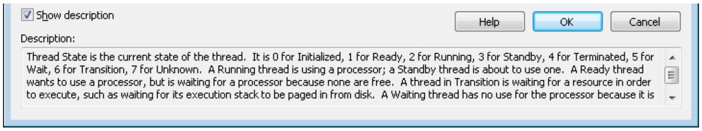


FIGURE 5-16 Simplified version of thread states and transitions

EXPERIMENT: Thread-Scheduling State Changes

You can watch thread-scheduling state changes with the Performance tool in Windows. This utility can be useful when you're debugging a multithreaded application and you're unsure about the state of the threads running in the process. To watch thread-scheduling state changes by using the Performance tool, follow these steps:

1. Run Notepad (Notepad.exe).
2. Start the Performance tool by selecting All Programs from the Start menu and then selecting Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
3. Select the chart view if you're in some other view.
4. Right-click on the graph, and choose Properties.
5. Click on the Graph tab, and change the chart vertical scale maximum to 7. (As you'll see from the explanation text for the performance counter, thread states are numbered from 0 through 7.) Click OK.
6. Click the Add button on the toolbar to bring up the Add Counters dialog box.
7. Select the Thread performance object, and then select the Thread State counter. Select the Show Description check box to see the definition of the values:

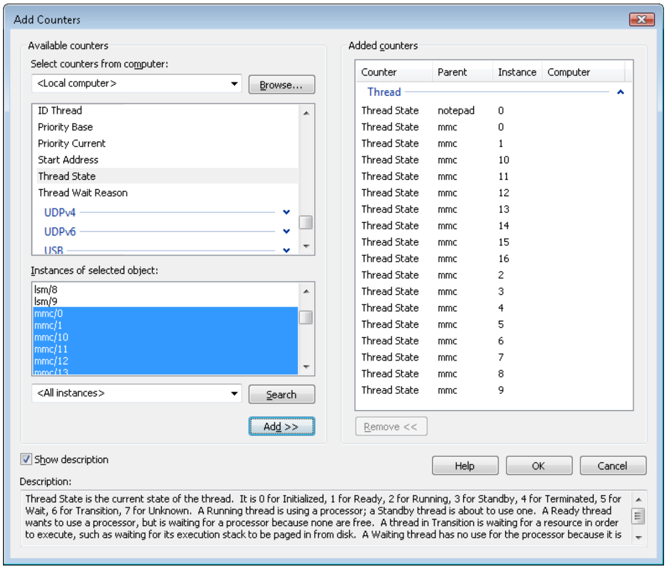


☑ Show description

Description:

Thread State is the current state of the thread. It is 0 for Initialized, 1 for Ready, 2 for Running, 3 for Standby, 4 for Terminated, 5 for Wait, 6 for Transition, 7 for Unknown. A Running thread is using a processor; a Standby thread is about to use one. A Ready thread wants to use a processor, but is waiting for a processor because none are free. A thread in Transition is waiting for a resource in order to execute, such as waiting for its execution stack to be paged in from disk. A Waiting thread has no use for the processor because it is

8. In the Instances box, select <All instances> and type Notepad before clicking Search. Scroll down until you see the Notepad process (*notepad/0*); select it, and click the Add button.
9. Scroll back up in the Instances box to the *Mmc* process (the Microsoft Management Console process running the System Monitor), select all the threads (*mmc/0*, *mmc/1*, and so on), and add them to the chart by clicking the Add button. Before you click Add, you should see something like the dialog box that follows.



Counter	Parent	Instance	Computer
Thread			
Thread State	notepad	0	
Thread State	mmc	0	
Thread State	mmc	1	
Thread State	mmc	10	
Thread State	mmc	11	
Thread State	mmc	12	
Thread State	mmc	13	
Thread State	mmc	14	
Thread State	mmc	15	
Thread State	mmc	16	
Thread State	mmc	2	
Thread State	mmc	3	
Thread State	mmc	4	
Thread State	mmc	5	
Thread State	mmc	6	
Thread State	mmc	7	
Thread State	mmc	8	
Thread State	mmc	9	

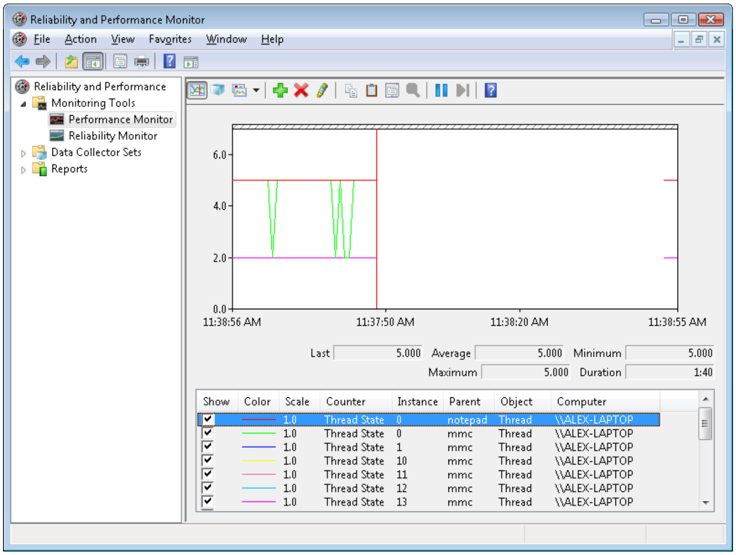
☑ Show description

Description:

Thread State is the current state of the thread. It is 0 for Initialized, 1 for Ready, 2 for Running, 3 for Standby, 4 for Terminated, 5 for Wait, 6 for Transition, 7 for Unknown. A Running thread is using a processor; a Standby thread is about to use one. A Ready thread wants to use a processor, but is waiting for a processor because none are free. A thread in Transition is waiting for a resource in order to execute, such as waiting for its execution stack to be paged in from disk. A Waiting thread has no use for the processor because it is

10. Now close the Add Counters dialog box by clicking OK.

11. You should see the state of the Notepad thread (the very top line in the following figure) as a 5. As shown in the explanation text you saw under step 7, this number represents the waiting state (because the thread is waiting for GUI input):



Last 5.000 Average 5.000 Minimum 5.000
Maximum 5.000 Duration 1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
☑	—	1.0	Thread State	0	notepad	Thread	\\ALEX-LAPTOP
☑	—	1.0	Thread State	0	mmc	Thread	\\ALEX-LAPTOP
☑	—	1.0	Thread State	1	mmc	Thread	\\ALEX-LAPTOP
☑	—	1.0	Thread State	10	mmc	Thread	\\ALEX-LAPTOP
☑	—	1.0	Thread State	11	mmc	Thread	\\ALEX-LAPTOP
☑	—	1.0	Thread State	12	mmc	Thread	\\ALEX-LAPTOP
☑	—	1.0	Thread State	13	mmc	Thread	\\ALEX-LAPTOP

12. Notice that one thread in the *Mmc* process (running the Performance tool snap-in) is in the running state (number 2). This is the thread that's querying the thread states, so it's always displayed in the running state.
13. You'll never see Notepad in the running state (unless you're on a multiprocessor system) because *Mmc* is always in the running state when it gathers the state of the threads you're monitoring.

Dispatcher Database

To make thread-scheduling decisions, the kernel maintains a set of data structures known collectively as the dispatcher database, illustrated in Figure 5-17. The dispatcher database keeps track of which threads are waiting to execute and which processors are executing which threads.

To improve scalability, including thread-dispatching concurrency, Windows multiprocessor systems have per-processor dispatcher ready queues, as illustrated in Figure 5-17. In this way, each CPU can check its own ready queues for the next thread to run without having to lock the systemwide ready queues.

The per-processor ready queues, as well as the per-processor ready summary, are part of the processor control block (PRCB) structure. (To see the fields in the PRCB, type **dt nt!_kprcb** in the kernel debugger.) The names of each component that we will talk about (in italics) are field members of the PRCB structure.

The dispatcher ready queues (*DispatcherReadyListHead*) contain the threads that are in the ready state, waiting to be scheduled for execution. There is one queue for each of the 32 priority levels. To speed up the selection of which thread to run or preempt, Windows maintains a 32-bit bit mask called the ready summary (*ReadySummary*). Each bit set indicates one or more threads in the ready queue for that priority level. (Bit 0 represents priority 0, and so on.)

Instead of scanning each ready list to see whether it is empty or not (which would make scheduling decisions dependent on the number of different priority threads), a single bit scan is performed as a native processor command to find the highest bit set. Regardless of the number of threads in the ready queue, this operation takes a constant amount of time, which is why you might sometimes see the Windows scheduling algorithm referred to as an O(1), or constant time, algorithm.

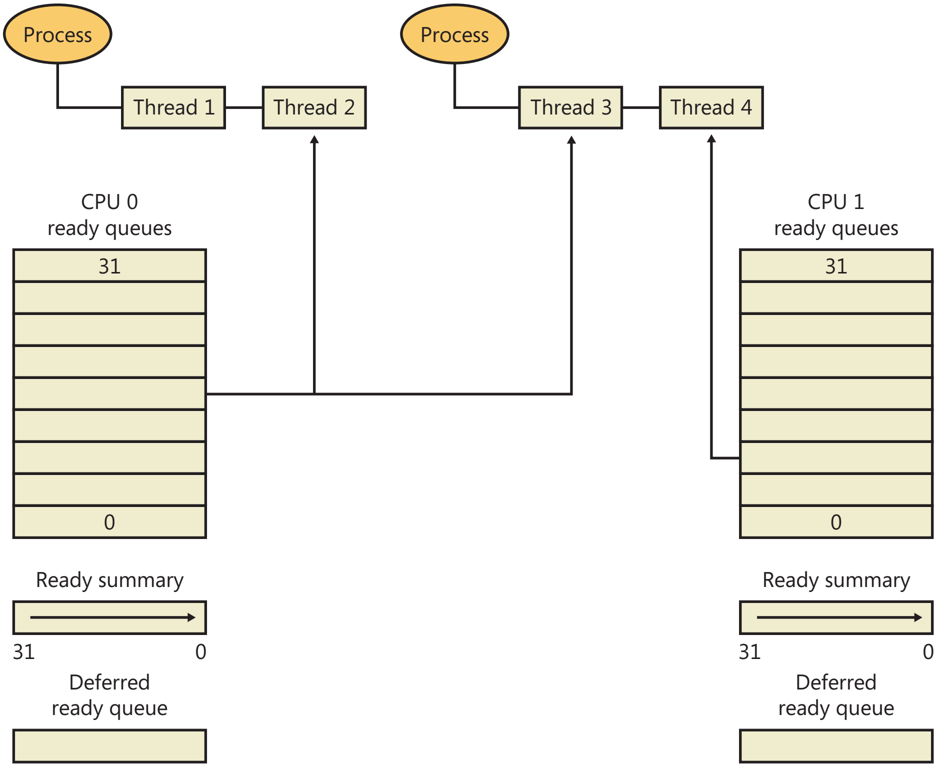


FIGURE 5-17 Windows multiprocessor dispatcher database

The dispatcher database is synchronized by raising IRQL to DISPATCH_LEVEL. (For an explanation of interrupt priority levels, see the "Trap Dispatching" section in Chapter 3.) Raising IRQL in this way prevents other threads from interrupting thread dispatching on the processor because threads normally run at IRQL 0 or 1. However, more is required than just raising IRQL, because other processors can simultaneously raise to the same IRQL and attempt to operate on their dispatcher database. How Windows synchronizes access to the dispatcher database is explained in the "Multiprocessor Systems" section later in the chapter.

Quantum

As mentioned earlier in the chapter, a quantum is the amount of time a thread gets to run before Windows checks to see whether another thread at the same priority is waiting to run. If a thread completes its quantum and there are no other threads at its priority, Windows permits the thread to run for another quantum.

On client versions of Windows, threads run by default for 2 clock intervals; on server systems, by default, a thread runs for 12 clock intervals. (We'll explain how you can change these values later.) The rationale for the longer default value on server systems is to minimize context switching. By having a longer quantum, server applications that wake up as the result of a client request have a better chance of completing the request and going back into a wait state before their quantum ends.

The length of the clock interval varies according to the hardware platform. The frequency of the clock interrupts is up to the HAL, not the kernel. For example, the clock interval for most x86 uniprocessors is about 10 milliseconds (note that these machines are no longer supported by Windows and are only used here for example purposes), and for most x86 and x64 multiprocessors it is about 15 milliseconds. This clock interval is stored in the kernel variable *KeMaximumIncrement* as hundreds of nanoseconds.

Because thread run-time accounting is based on processor cycles, although threads still run in units of clock intervals, the system does not use the count of clock ticks as the deciding factor for how long a thread has run and whether its quantum has expired. Instead, when the system starts up, a calculation is made whose result is the number of clock cycles that each quantum is equivalent to. (This value is stored in the kernel variable *KiCyclesPerClockQuantum*.) This calculation is made by multiplying the processor speed in Hz (CPU clock cycles per second) with the number of seconds it takes for one clock tick to fire (based on the *KeMaximumIncrement* value described earlier).

The result of this accounting method is that threads do not actually run for a quantum number based on clock ticks; they instead run for a quantum target, which represents an estimate of what the number of CPU clock cycles the thread has consumed should be when its turn would be given up. This target should be equal to an equivalent number of clock interval timer ticks because, as you just saw, the calculation of clock cycles per quantum is based on the clock interval timer frequency, which you can check using the following experiment. On the other hand, because interrupt cycles are not charged to the thread, the actual clock time might be longer.

EXPERIMENT: Determining the Clock Interval Frequency

The Windows *GetSystemTimeAdjustment* function returns the clock interval. To determine the clock interval, download and run the Clockres program from Windows Sysinternals (www.microsoft.com/technet/sysinternals). Here's the output from a dual-core 64-bit Windows 7 system:

```
C:\>clockres

ClockRes v2.0 - View the system clock resolution
Copyright (C) 2009 Mark Russinovich
SysInternals - www.sysinternals.com

Maximum timer interval: 15.600 ms
Minimum timer interval: 0.500 ms
Current timer interval: 15.600 ms
```

Quantum Accounting

Each process has a quantum reset value in the process control block (KPROCESS). This value is used when creating new threads inside the process and is duplicated in the thread control block (KTHREAD), which is then used when giving a thread a new quantum target. The quantum reset value is stored in terms of actual quantum units (we'll discuss what these mean soon), which are then multiplied by the number of clock cycles per quantum, resulting in the quantum target.

As a thread runs, CPU clock cycles are charged at different events (context switches, interrupts, and certain scheduling decisions). If at a clock interval timer interrupt, the number of CPU clock cycles charged has reached (or passed) the quantum target, quantum end processing is triggered. If there is another thread at the same priority waiting to run, a context switch occurs to the next thread in the ready queue.

Internally, a quantum unit is represented as one third of a clock tick. (So one clock tick equals three quantums.) This means that on client Windows systems, threads, by default, have a quantum reset value of *6 (2 * 3)*, and that server systems have a quantum reset value of 36 (12 * 3). For this reason, the *KiCyclesPerClockQuantum* value is divided by three at the end of the calculation previously described, because the original value describes only CPU clock cycles per clock interval timer tick.

The reason a quantum was stored internally as a fraction of a clock tick rather than as an entire tick was to allow for partial quantum decay-on-wait completion on versions of Windows prior to Windows Vista. Prior versions used the clock interval timer for quantum expiration. If this adjustment were not made, it would have been possible for threads never to have their quantums reduced. For example, if a thread ran, entered a wait state, ran again, and entered another wait state but was never the currently running thread when the clock interval timer fired, it would never have its quantum charged for the time it was running. Because threads now have CPU clock cycles charged instead of quantums, and because this no longer depends on the clock interval timer, these adjustments are not required.

EXPERIMENT: Determining the Clock Cycles per Quantum

Windows doesn't expose the number of clock cycles per quantum through any function, but with the calculation and description we've given, you should be able to determine this on your own using the following steps and a kernel debugger such as WinDbg in local debugging mode:

1. Obtain your processor frequency as Windows has detected it. You can use the value stored in the PRCB's MHz field, which can be displayed with the *!cpuinfo* command. Here is a sample output of a dual-core Intel system running at 2829 MHz:

```
lkd> !cpuinfo
CP  F/M/S Manufacturer  MHz PRCB Signature    MSR 8B Signature Features
 0  6,15,6 GenuineIntel 2829 000000c700000000 >000000c700000000<a00f3fff
 1  6,15,6 GenuineIntel 2829 000000c700000000                   a00f3fff
                      Cached Update Signature 000000c700000000
                     Initial Update Signature 000000c700000000
```

2. Convert the number to Hertz (Hz). This is the number of CPU clock cycles that occur each second on your system. In this case, 2,829,000,000 cycles per second.

3. Obtain the clock interval on your system by using *clockres*. This measures how long it takes before the clock fires. On the sample system used here, this interval was 15.600100 ms.

4. Convert this number to the number of times the clock interval timer fires each second. One second is 1000 ms, so divide the number derived in step 3 by 1000. In this case, the timer fires every 0.0156001 seconds.

5. Multiply this count by the number of cycles each second that you obtained in step 2. In our case, 44,132,682.9 cycles have elapsed after each clock interval.

6. Remember that each quantum unit is one-third of a clock interval, so divide the number of cycles by three. In our example, this gives us 14,710,894, or 0xE0786E in hexadecimal. This is the number of clock cycles each quantum unit should take on a system running at 2829 MHz with a clock interval of around 15 ms.

7. To verify your calculation, dump the value of *KiCyclesPerClockQuantum* on your system—it should match.

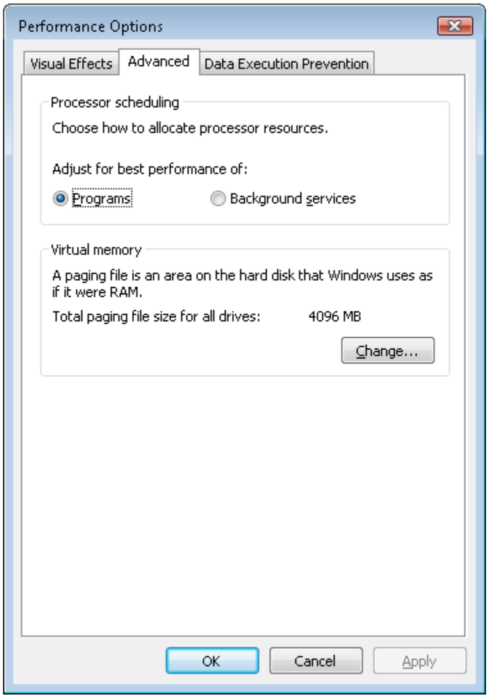
```
lkd> dd nt!KiCyclesPerClockQuantum L1
81d31ae8  00e0786e
```

Controlling the Quantum

You can change the thread quantum for all processes, but you can choose only one of two settings: short (2 clock ticks, which is the default for client machines) or long (12 clock ticks, which is the default for server systems).

Note By using the job object on a system running with long quantums, you can select other quantum values for the processes in the job. For more information on the job object, see the “Job Objects” section later in the chapter.

To change this setting, right-click on your Computer icon on the desktop, or in Windows Explorer, choose Properties, click the Advanced System Settings label, click on the Advanced tab, click the Settings button in the Performance section, and finally click on the Advanced tab. The dialog box displayed is shown in Figure 5-18.



Processor scheduling

Choose how to allocate processor resources.

Adjust for best performance of:

● Programs ○ Background services

Virtual memory

A paging file is an area on the hard disk that Windows uses as if it were RAM.

Total paging file size for all drives: 4096 MB

FIGURE 5-18 Quantum configuration in the Performance Options dialog box

The Programs setting designates the use of short, variable quantums—the default for client versions of Windows. If you install Terminal Services on a server system and configure the server as an application server, this setting is selected so that the users on the terminal server have the same quantum settings that would normally be set on a desktop or client system. You might also select this manually if you were running Windows Server as your desktop operating system.

The Background Services option designates the use of long, fixed quantums—the default for server systems. The only reason you might select this option on a workstation system is if you were using the workstation as a server system. However, because changes in this option take effect immediately, it might make sense to use it if the machine is about to run a background/server-style workload. For example, if a long-running computation, encoding or modeling simulation needs to run overnight, Background Services mode could be selected at night, and the system put back in Programs mode in the morning.

Finally, because Programs mode enables variable quantums, let us now explain what controls their variability.

Variable Quantums

When variable quantums are enabled, the variable quantum table (*PspVariableQuantums*) is loaded into the *PspForegroundQuantum* table that is used by the *PspComputeQuantum* function. Its algorithm will pick the appropriate quantum index based on whether or not the process is a foreground process (that is, whether it contains the thread that owns the foreground window on the desktop). If this is not the case, an index of zero is chosen, which corresponds to the default thread quantum described earlier. If it is a foreground process, the quantum index corresponds to the priority separation.

This priority separation value determines the priority boost (described in a later section of this chapter) that the scheduler will apply to foreground threads, and it is thus paired with an appropriate extension of the quantum: for each extra priority level (up to 2), another quantum is given to the thread. For example, if the thread receives a boost of one priority level, it receives an extra quantum as well. By default, Windows sets the maximum possible priority boost to foreground threads, meaning that the priority separation will be 2, therefore selecting quantum index 2 in the variable quantum table, leading to the thread receiving two extra quantums, for a total of 3 quantums.

Table 5-5 describes the exact quantum value (recall that this is stored in a unit representing 1/3rd of a clock tick) that will be selected based on the quantum index and which quantum configuration is in use.

TABLE 5-5 Quantum Values

	Short Quantum Index			**Long Quantum Index**		
Variable	6	12	18	12	24	36
Fixed	18	18	18	36	36	36

Thus, when a window is brought into the foreground on a client system, all the threads in the process containing the thread that owns the foreground window have their quantums tripled: threads in the foreground process run with a quantum of 6 clock ticks, whereas threads in other processes have the default client quantum of 2 clock ticks. In this way, when you switch away from a CPU-intensive process, the new foreground process will get proportionally more of the CPU, because when its threads run they will have a longer turn than background threads (again, assuming the thread priorities are the same in both the foreground and background processes).

Quantum Settings Registry Value

The user interface to control quantum settings described earlier modifies the registry value HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation. In addition to specifying the relative length of thread quantums (short or long), this registry value also defines whether or not variable quantums should be used, as well as the priority separation (which, as you've seen, will determine the quantum index used when variable quantums are enabled). This value consists of 6 bits divided into the three 2-bit fields shown in Figure 5-19.

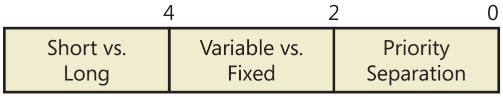


FIGURE 5-19 Fields of the Win32PrioritySeparation registry value

The fields shown in Figure 5-19 can be defined as follows:

- **Short vs. Long** A value of 1 specifies long quantums, and 2 specifies short ones. A setting of 0 or 3 indicates that the default appropriate for the system will be used (short for client systems, long for server systems).
- **Variable vs. Fixed** A setting of 1 means to enable the variable quantum table based on the algorithm shown in the "Variable Quantums" section. A setting of *0* or *3* means that the default appropriate for the system will be used (variable for client systems, fixed for server systems).
- **Priority Separation** This field (stored in the kernel variable *PsPrioritySeparation*) defines the priority separation (up to 2) as explained in the "Variable Quantums" section.

Note that when you're using the Performance Options dialog box (which was shown in Figure 5-18), you can choose from only two combinations: short quantums with foreground quantums tripled, or long quantums with no quantum changes for foreground threads. However, you can select other combinations by modifying the *Win32PrioritySeparation* registry value directly.

Note that the threads part of a process running in the idle process priority class always receive a single thread quantum (2 clock ticks), ignoring any sort of quantum configuration settings, whether set by default or set through the registry.

On Windows Server systems configured as applications servers, the initial value of the *Win32PrioritySeparation* registry value will be hex 26, which is identical to the value set by the Optimize Performance For Programs option in the Performance Options dialog box. This selects quantum and priority boost behavior like that on Windows client systems, which is appropriate for a server primarily used to host users' applications.

On Windows client systems and on servers not configured as application servers, the initial value of the *Win32PrioritySeparation* registry value will be 2. This provides values of 0 for the Short vs. Long and Variable vs. Fixed bit fields, relying on the default behavior of the system (depending on whether it is a client system or a server system) for these options, but it provides a value of 2 for the Priority Separation field. Once the registry value has been changed by use of the Performance Options dialog box, it cannot be restored to this original value other than by modifying the registry directly.

EXPERIMENT: Effects of Changing the Quantum Configuration

Using a local debugger (Kd or WinDbg), you can see how the two quantum configuration settings, Programs and Background Services, affect the *PsPrioritySeparation* and *PspForegroundQuantum* tables, as well as modify the *QuantumReset* value of threads on the system. Take the following steps:

1. Open the System utility in Control Panel (or right-click on your computer name's icon on the desktop, and choose Properties). Click the Advanced System Settings label, click on the Advanced tab, click the Settings button in the Performance section, and finally click on the Advanced tab. Select the Programs option, and click Apply. Keep this window open for the duration of the experiment.

2. Dump the values of *PsPrioritySeparation* and *PspForegroundQuantum*, as shown here. The values shown are what you should see on a Windows system after making the change in step 1. Notice how the variable, short quantum table is being used, and that a priority boost of 2 will apply to foreground applications:

```
lkd> dd PsPrioritySeparation L1
81d3101c  00000002
lkd> db PspForegroundQuantum L3
81d0946c  06 0c 12
...
```

3. Now take a look at the *QuantumReset* value of any process on the system. As described earlier, this is the default, full quantum of each thread on the system when it is replenished. This value is cached into each thread of the process, but the KPROCESS structure is easier to look at. Notice in this case it is 6, because WinDbg, like most other applications, gets the quantum set in the first entry of the *PspForegroundQuantum* table:

```
lkd> .process
Implicit process is now 85b32d90
lkd> dt nt!_KPROCESS 85b32d90 QuantumReset
nt!_KPROCESS
   +0x061 QuantumReset     : 6 ''
```

4. Now change the Performance option to Background Services in the dialog box you opened in step 1.

5. Repeat the commands shown in steps 2 and 3. You should see the values change in a manner consistent with our discussion in this section:

```
lkd> dd nt!PsPrioritySeparation L1
81d3101c  00000000
lkd> db nt!PspForegroundQuantum L3
81d0946c  24 24 24                                         $$$
lkd> dt nt!_KPROCESS 85b32d90 QuantumReset
nt!_KPROCESS
   +0x061 QuantumReset     : 36 '$'
```

Priority Boosts

The Windows scheduler periodically adjusts the current priority of threads through an internal priority-boosting mechanism. In many cases, it does so for decreasing various latencies (that is, to make threads respond faster to the events they are waiting on) and increasing responsiveness. In others, it applies these boosts to prevent inversion and starvation scenarios. Here are some of the boost scenarios that will be described in this section (and their purpose):

- Boosts due to scheduler/dispatcher events (latency reduction)
- Boosts due to I/O completion (latency reduction)
- Boosts due to UI input (latency reduction/responsiveness)
- Boosts due to a thread waiting on an executive resource for too long (starvation avoidance)
- Boosts when a thread that's ready to run hasn't been running for some time (starvation and priority-inversion avoidance)

Like any scheduling algorithms, however, these adjustments aren't perfect, and they might not benefit all applications.

> **Note** Windows never boosts the priority of threads in the real-time range (16 through 31). Therefore, scheduling is always predictable with respect to other threads in the real-time range. Windows assumes that if you're using the real-time thread priorities, you know what you're doing.

Client versions of Windows also include another pseudo-boosting mechanism that occurs during multimedia playback. Unlike the other priority boosts, which are applied directly by kernel code, multimedia playback boosts are actually managed by a user-mode service called the MultiMedia Class Scheduler Service (MMCSS), but they are not really boosts—the service merely sets new base priorities for the threads as needed (by calling the user-mode native API to change thread priorities). Therefore, none of the rules regarding boosts apply. We'll first cover the typical kernel-managed priority boosts and then talk about MMCSS and the kind of "boosting" it performs.

Boosts Due to Scheduler/Dispatcher Events

Whenever a dispatch event occurs, the *KiExitDispatcher* routine is called, whose job it is to process the deferred ready list by calling *KiProcessThreadWaitList* and then call *KiCheckForThreadDispatch* to check whether any threads on the local processor should not be scheduled. Whenever such an event occurs, the caller can also specify which type of boost should be applied to the thread, as well as what priority increment the boost should be associated with. The following scenarios are considered as *AdjustUnwait* dispatch events because they deal with a dispatcher object entering a signaled state, which might cause one or more threads to wake up:

- An APC is queued to a thread.
- An event is set or pulsed.

- A timer was set, or the system time was changed, and timers had to be reset.
- A mutex was released or abandoned.
- A process exited.
- An entry was inserted in a queue, or the queue was flushed.
- A semaphore was released.
- A thread was alerted, suspended, resumed, frozen, or thawed.
- A primary UMS thread is waiting to switch to a scheduled UMS thread.

For scheduling events associated with a public API (such as *SetEvent*), the boost increment applied is specified by the caller. Windows recommends certain values to be used by developers, which will be described later. For alerts, a boost of 2 is applied, because the alert API does not have a parameter allowing a caller to set a custom increment.

The scheduler also has two special *AdjustBoost* dispatch events, which are part of the lock ownership priority mechanism. These boosts attempt to fix situations in which a caller that owns the lock at priority X ends up releasing the lock to a waiting thread at priority <= X. In this situation, the new owner thread must wait for its turn (if running at priority X), or worse, it might not even get to run at all if its priority is lower than X. This entails the releasing thread continuing its execution, even though it should have caused the new owner thread to wake up and take control of the processor. The following two dispatcher events cause an *AdjustBoost* dispatcher exit:

- An event is set through the *KeSetEventBoostPriority* interface, which is used by the ERESOURCE reader-writer kernel lock
- A gate is set through the *KeSignalGateBoostPriority* interface, which is used by various internal mechanisms when releasing a gate lock.

Unwait Boosts

Unwait boosts attempt to decrease the latency between a thread waking up due to an object being signaled (thus entering the Ready state) and the thread actually beginning its execution to process the unwait (thus entering the Running state). Because the event that the thread is waiting on could give some sort of information about, say, the state of available memory at the moment, it is important for this state not to change behind the scenes while the thread is still stuck in the Ready state—otherwise, it might become irrelevant or incorrect once the thread does start running.

The various Windows header files specify recommended values that kernel-mode callers of APIs such as *KeSetEvent* and *KeReleaseSemaphore* should use, which correspond to definitions such as MUTANT_INCREMENT and EVENT_INCREMENT. These definitions have always been set to 1 in the headers, so it is safe to assume that most unwaits on these objects result in a boost of 1. In the user-mode API, an increment cannot be specified, nor do the native system calls such as *NtSetEvent* have parameters to specify such a boost. Instead, when these APIs call the underlying *Ke* interface, they use the default _INCREMENT definition automatically. This is also the case when mutexes are abandoned or timers are reset due to a system time change: the system uses the default boost that normally would've been applied when the mutex would have been released. Finally, the APC boost is completely up to the caller. Soon, you'll see a specific usage of the APC boost related to I/O completion.

Note Some dispatcher objects don't have boosts associated with them. For example, when a timer is set or expires, or when a process is signaled, no boost is applied.

All these boosts of +1 attempt to solve the initial problem by making the assumption that both the releasing and waiting threads are running at the same priority. By boosting the waiting thread by one priority level, the waiting thread should preempt the releasing thread as soon as the operation completes. Unfortunately on uniprocessor systems, if this assumption does not hold, the boost might not do much: if the waiting thread is waiting at priority 4 vs. the releasing thread at priority 8, waiting at priority 5 won't do much to reduce latency and force preemption. On multiprocessor systems, however, due to the stealing and balancing algorithms, this higher priority thread may have a higher chance to get picked up by another logical processor. This reality is due to a design choice made in the initial NT architecture, which is not to track lock ownership (except a few locks). That means the scheduler can't be sure who really owns an event, and if it's really being used as a lock. Even with lock ownership tracking, ownership is not usually passed in order to avoid convoy issues, other than in the ERESOURCE case which we'll explain below.

However, for certain kinds of lock objects using events or gates as their underlying synchronization object, the lock ownership boost resolves the dilemma. Also, due to the processor-distribution and load-balancing schemes you'll see later, on a multiprocessor machine, the ready thread might get picked up on another processor, and its high priority might increase the chances of it running on that secondary processor instead.

Lock Ownership Boosts

Because the executive-resource (ERESOURCE) and critical-section locks use underlying dispatcher objects, releasing these locks results in an unwait boost as described earlier. On the other hand, because the high-level implementation of these objects does track the owner of the lock, the kernel can make a more informed decision as to what kind of boost should be applied, by using the *AdjustBoost* reason. In these kinds of boosts, *AdjustIncrement* is set to the current priority of the releasing (or setting) thread, minus any GUI foreground separation boost, and before the *KiExitDispatcher* function is called, *KiRemoveBoostThread* is called by the event and gate code to return the releasing thread back to its regular priority (through the *KiComputeNewPriority* function). This step is needed to avoid a lock convoy situation, in which two threads repeatedly passing the lock between one another get ever-increasing boosts.

Note that pushlocks, which are unfair locks because ownership of the lock in a contended acquisition path is not predictable (rather, it's random, just like a spinlock), do not apply priority boosts due to lock ownership. This is because doing so only contributes to preemption and priority proliferation, which isn't required because the lock becomes immediately free as soon as it is released (bypassing the normal wait/unwait path).

Other differences between the lock ownership boost and the unwait boost will be exposed in the way that the scheduler actually applies boosting, which is the upcoming topic after this section.

Priority Boosting After I/O Completion

Windows gives temporary priority boosts upon completion of certain I/O operations so that threads that were waiting for an I/O have more of a chance to run right away and process whatever was being waited for. Although you'll find recommended boost values in the Windows Driver Kit (WDK) header files (by searching for "#define IO" in Wdm.h or Ntddk.h), the actual value for the boost is up to the device driver. (These values are listed in Table 5-6.) It is the device driver that specifies the boost when it completes an I/O request on its call to the kernel function, *IoCompleteRequest*. In Table 5-6, notice that I/O requests to devices that warrant better responsiveness have higher boost values.

TABLE 5-6 Recommended Boost Values

Device	Boost
Disk, CD-ROM, parallel, video	1
Network, mailslot, named pipe, serial	2
Keyboard, mouse	6
Sound	8

Note You might intuitively expect "better responsiveness" from your video card or disk than a boost of 1, but in fact, the kernel is trying to optimize for *latency*, which some devices (as well as human sensory inputs) are more sensitive to than others. To give you an idea, a sound card expects data around every 1 ms to play back music without perceptible glitches, while a video card needs to output at only 24 frames per second, or once every 40 ms, before the human eye can notice glitches.

As hinted earlier, these I/O completion boosts rely on the unwait boosts seen in the previous section. In Chapter 8 of Part 2, the mechanism of I/O completion will be shown in depth. For now, the important detail is that the kernel implements the signaling code in the *IoCompleteRequest* API through the use of either an APC (for asynchronous I/O) or through an event (for synchronous I/O). When a driver passes in, for example, IO_DISK_INCREMENT to *IoCompleteRequest* for an asynchronous disk read, the kernel calls *KeInsertQueueApc* with the boost parameter set to IO_DISK_INCREMENT. In turn, when the thread's wait is broken due to the APC, it receives a boost of *1*.

Be aware that the boost values given in the previous table are merely recommendations by Microsoft—driver developers are free to ignore them if they choose to do so, and certain specialized drivers can use their own values. For example, a driver handling ultrasound data from a medical device, which must notify a user-mode visualization application of new data, would probably use a boost value of *8* as well, to satisfy the same latency as a sound card.

In most cases, however, due to the way Windows driver stacks are built (again, see Chapter 8, "I/O System," in Part 2 for more information), driver developers often write *minidrivers*, which call into a Microsoft-owned driver that supplies its own boost to *IoCompleteRequest*. For example, RAID or SATA controller card developers would typically call *StorPortCompleteRequest* to complete processing their requests. This call does not have any parameter for a boost value, because the Storport.sys driver fills in the right value when calling the kernel.

Additionally, in newer versions of Windows, whenever any file system driver (identified by setting its device type to FILE_DEVICE_DISK_FILE_SYSTEM or FILE_DEVICE_NETWORK_FILE_SYSTEM) completes its request, a boost of IO_DISK_INCREMENT is always applied if the driver passed in IO_NO_INCREMENT instead. So this boost value has become less of a recommendation and more of a requirement enforced by the kernel.

Boosts During Waiting on Executive Resources

When a thread attempts to acquire an executive resource (ERESOURCE; see Chapter 3 for more information on kernel-synchronization objects) that is already owned exclusively by another thread, it must enter a wait state until the other thread has released the resource. To limit the risk of deadlocks, the executive performs this wait in intervals of five seconds instead of doing an infinite wait on the resource.

At the end of these five seconds, if the resource is still owned, the executive attempts to prevent CPU starvation by acquiring the dispatcher lock, boosting the owning thread or threads to 14 (only if the original owner priority is less than the waiter's and not already 14), resetting their quantums, and performing another wait.

Because executive resources can be either shared or exclusive, the kernel first boosts the exclusive owner and then checks for shared owners and boosts all of them. When the waiting thread enters the wait state again, the hope is that the scheduler will schedule one of the owner threads, which will have enough time to complete its work and release the resource. Note that this boosting mechanism is used only if the resource doesn't have the Disable Boost flag set, which developers can choose to set if the priority-inversion mechanism described here works well with their usage of the resource.

Additionally, this mechanism isn't perfect. For example, if the resource has multiple shared owners, the executive boosts all those threads to priority 14, resulting in a sudden surge of high-priority threads on the system, all with full quantums. Although the initial owner thread will run first (because it was the first to be boosted and therefore is first on the ready list), the other shared owners will run next, because the waiting thread's priority was not boosted. Only after all the shared owners have had a chance to run and their priority has been decreased below the waiting thread will the waiting thread finally get its chance to acquire the resource. Because shared owners can promote or convert their ownership from shared to exclusive as soon as the exclusive owner releases the resource, it's possible for this mechanism not to work as intended.

Priority Boosts for Foreground Threads After Waits

As will be shortly described, whenever a thread in the foreground process completes a wait operation on a kernel object, the kernel boosts its current (not base) priority by the current value of *PsPrioritySeparation*. (The windowing system is responsible for determining which process is considered to be in the foreground.) As described in the section on quantum controls, *PsPrioritySeparation* reflects the quantum-table index used to select quantums for the threads of foreground applications. However, in this case, it is being used as a priority boost value.

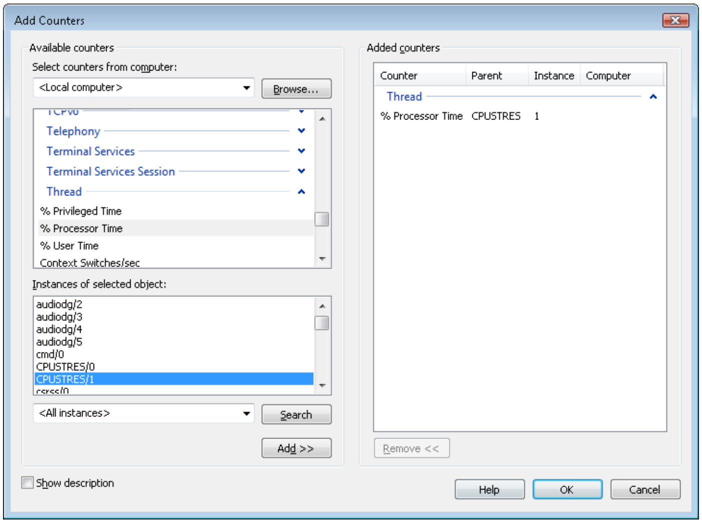
The reason for this boost is to improve the responsiveness of interactive applications—by giving the foreground application a small boost when it completes a wait, it has a better chance of running right away, especially when other processes at the same base priority might be running in the background.

EXPERIMENT: Watching Foreground Priority Boosts and Decays

Using the CPU Stress tool (downloadable from *http://live.sysinternals.com/WindowsInternals)*, you can watch priority boosts in action. Take the following steps:

1. Open the System utility in Control Panel (or right-click on your computer name's icon on the desktop, and choose Properties). Click the Advanced System Settings label, click on the Advanced tab, click the Settings button in the Performance section, and finally click on the Advanced tab. Select the Programs option. This causes *PsPrioritySeparation* to get a value of *2*.
2. Run Cpustres.exe, and change the activity of thread 1 from Low to Busy.
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add Counters dialog box.
5. Select the Thread object, and then select the % Processor Time counter.

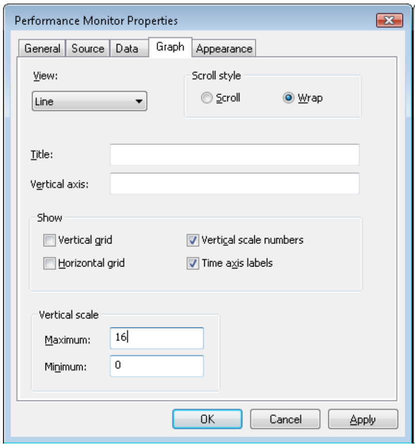
6. In the Instances box, select <All Instances> and click Search. Scroll down until you see the CPUSTRES process. Select the second thread (thread 1). (The first thread is the GUI thread.) You should see something like this:



Counter	Parent	Instance	Computer
Thread			
% Processor Time	CPUSTRES	1	

7. Click the Add button, and then click OK.

8. Select Properties from the Action menu. Change the Vertical Scale Maximum to *16* on the Graph tab, and set the interval to *1* in Sample Every box of the Graph Elements area on the General tab.



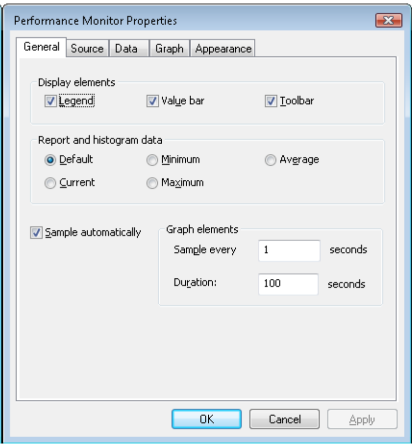
Scroll style: ○ Scroll ● Wrap

☐ Vertical grid ☑ Vertical scale numbers

☐ Horizontal grid ☑ Time axis labels

Maximum: 16

Minimum: 0



☑ Legend ☑ Value bar ☑ Toolbar

● Default ○ Minimum ○ Average

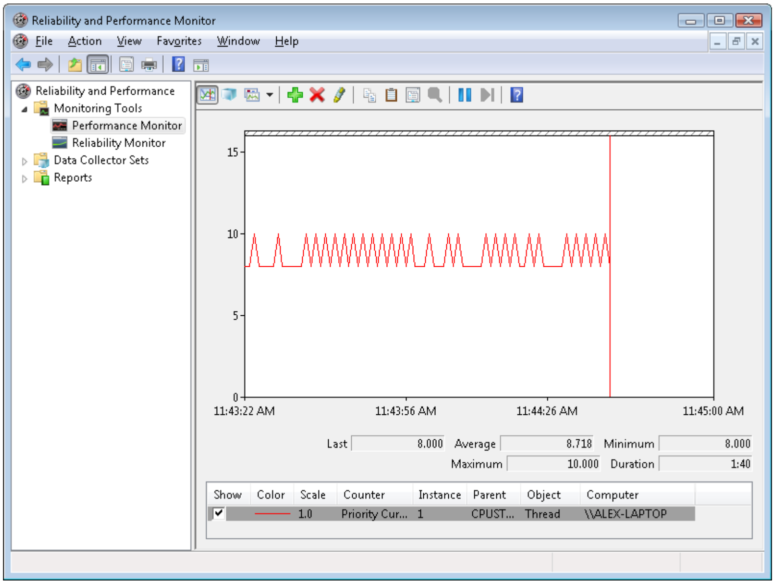
○ Current ○ Maximum

☑ Sample automatically

Sample every: 1 seconds

Duration: 100 seconds

9. Now bring the CPUSTRES process to the foreground. You should see the priority of the CPUSTRES thread being boosted by 2 and then decaying back to the base priority as follows:



Last 8.000 Average 8.718 Minimum 8.000
Maximum 10.000 Duration 1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
☑	—	1.0	Priority Cur...	1	CPUST...	Thread	\\ALEX-LAPTOP

10. The reason CPUSTRES receives a boost of 2 periodically is because the thread you're monitoring is sleeping about 25 percent of the time and then waking up. (This is the Busy Activity level). The boost is applied when the thread wakes up. If you set the Activity level to Maximum, you won't see any boosts because Maximum in CPUSTRES puts the thread into an infinite loop. Therefore, the thread doesn't invoke any wait functions and, as a result, doesn't receive any boosts.

11. When you've finished, exit Performance Monitor and CPU Stress.

Priority Boosts After GUI Threads Wake Up

Threads that own windows receive an additional boost of 2 when they wake up because of windowing activity such as the arrival of window messages. The windowing system (Win32k.sys) applies this boost when it calls *KeSetEvent* to set an event used to wake up a GUI thread. The reason for this boost is similar to the previous one—to favor interactive applications.

EXPERIMENT: Watching Priority Boosts on GUI Threads

You can also see the windowing system apply its boost of 2 for GUI threads that wake up to process window messages by monitoring the current priority of a GUI application and moving the mouse across the window. Just follow these steps:

1. Open the System utility in Control Panel (or right-click on your computer name's icon on the desktop, and choose Properties). Click the Advanced System Settings label, click on the Advanced tab, click the Settings button in the Performance section, and finally click on the Advanced tab. Be sure that the Programs option is selected. This causes *PsPrioritySeparation* to get a value of *2*.
2. Run Notepad from the Start menu by selecting All Programs/Accessories/Notepad.
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
4. Click the Add Counter toolbar button (or press Ctrl+N) to bring up the Add Counters dialog box.
5. Select the Thread object, and then select the Priority Current counter.
6. In the Instances box, type **Notepad**, and then click Search. Scroll down until you see Notepad/0. Click it, click the Add button, and then click OK.
7. As in the previous experiment, select Properties from the Action menu. Change the Vertical Scale Maximum to *16* on the Graph tab, set the interval to *1* in Sample Every box of the Graph Elements area of the General tab, and click OK.
8. You should see the priority of thread 0 in Notepad at 8 or 10. Because Notepad entered a wait state shortly after it received the boost of 2 that threads in the foreground process receive, it might not yet have decayed from 10 to 8.
9. With Performance Monitor in the foreground, move the mouse across the Notepad window. (Make both windows visible on the desktop.) You'll see that the priority sometimes remains at 10 and sometimes at 9, for the reasons just explained. (The reason you won't likely catch Notepad at 8 is that it runs so little after receiving the GUI thread boost of 2 that it never experiences more than one priority level of decay before waking up again because of additional windowing activity and receiving the boost of 2 again.)
10. Now bring Notepad to the foreground. You should see the priority rise to 12 and remain there (or drop to 11, because it might experience the normal priority decay that occurs for boosted threads on the quantum end) because the thread is receiving

two boosts: the boost of 2 applied to GUI threads when they wake up to process windowing input, and an additional boost of 2 because Notepad is in the foreground.

11. If you then move the mouse over Notepad (while it's still in the foreground), you might see the priority drop to 11 (or maybe even 10) as it experiences the priority decay that normally occurs on boosted threads as they complete their turn. However, the boost of 2 that is applied because it's the foreground process remains as long as Notepad remains in the foreground.
12. When you've finished, exit Performance Monitor and Notepad.

Priority Boosts for CPU Starvation

Imagine the following situation: you have a priority 7 thread that's running, preventing a priority 4 thread from ever receiving CPU time; however, a priority 11 thread is waiting for some resource that the priority 4 thread has locked. But because the priority 7 thread in the middle is eating up all the CPU time, the priority 4 thread will never run long enough to finish whatever it's doing and release the resource blocking the priority 11 thread. What does Windows do to address this situation?

You previously saw how the executive code responsible for executive resources manages this scenario by boosting the owner threads so that they can have a chance to run and release the resource. However, executive resources are only one of the many synchronization constructs available to developers, and the boosting technique will not apply to any other primitive. Therefore, Windows also includes a generic CPU starvation-relief mechanism as part of a thread called the balance set manager (a system thread that exists primarily to perform memory-management functions and is described in more detail in Chapter 10 of Part 2).

Once per second, this thread scans the ready queues for any threads that have been in the ready state (that is, haven't run) for approximately 4 seconds. If it finds such a thread, the balance-set manager boosts the thread's priority to 15 and sets the quantum target to an equivalent CPU clock cycle count of 3 quantum units. Once the quantum expires, the thread's priority decays immediately to its original base priority. If the thread wasn't finished and a higher priority thread is ready to run, the decayed thread returns to the ready queue, where it again becomes eligible for another boost if it remains there for another 4 seconds.

The balance-set manager doesn't actually scan all of the ready threads every time it runs. To minimize the CPU time it uses, it scans only 16 ready threads; if there are more threads at that priority level, it remembers where it left off and picks up again on the next pass. Also, it will boost only 10 threads per pass—if it finds 10 threads meriting this particular boost (which indicates an unusually busy system), it stops the scan at that point and picks up again on the next pass.

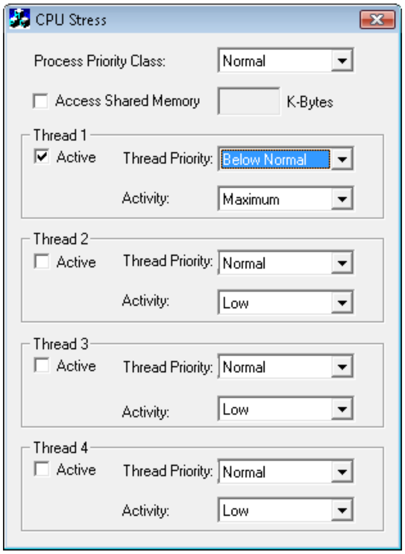
Note We mentioned earlier that scheduling decisions in Windows are not affected by the number of threads and that they are made in constant time, or O(1). Because the balance-set manager needs to scan ready queues manually, this operation depends on the number of threads on the system, and more threads will require more scanning time. However, the balance-set manager is not considered part of the scheduler or its algorithms and is simply an extended mechanism to increase reliability. Additionally, because of the cap on threads and queues to scan, the performance impact is minimized and predictable in a worst-case scenario.

Will this algorithm always solve the priority-inversion issue? No—it's not perfect by any means. But over time, CPU-starved threads should get enough CPU time to finish whatever processing they were doing and re-enter a wait state.

EXPERIMENT: Watching Priority Boosts for CPU Starvation

Using the CPU Stress tool, you can watch priority boosts in action. In this experiment, you'll see CPU usage change when a thread's priority is boosted. Take the following steps:

1. Run Cpustres.exe. Change the activity level of the active thread (by default, Thread 1) from Low to Maximum. Change the thread priority from Normal to Below Normal. The screen should look like this:



CPU Stress

Process Priority Class: Normal

☐ Access Shared Memory K-Bytes

Thread 1
☑ Active Thread Priority: Below Normal
Activity: Maximum

Thread 2
☐ Active Thread Priority: Normal
Activity: Low

Thread 3
☐ Active Thread Priority: Normal
Activity: Low

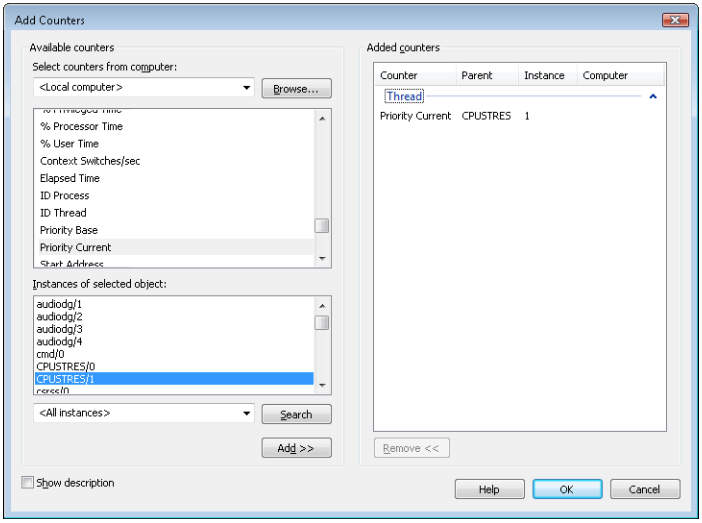
Thread 4
☐ Active Thread Priority: Normal
Activity: Low

2. Start the Performance tool by selecting Programs from the Start menu and then selecting Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.

3. Click the Add Counter toolbar button (or press Ctrl+N) to bring up the Add Counters dialog box.

4. Select the Thread object, and then select the Priority Current counter.

5. In the Instances box, type CPUSTRES, and then click Search. Scroll down until you see the second thread (thread 1). (The first thread is the GUI thread.) You should see something like this:



6. Click the Add button, and then click OK.

7. Raise the priority of Performance Monitor to real time by running Task Manager, clicking on the Processes tab, and selecting the Mmc.exe process. Right-click the process, select Set Priority, and then select Realtime. (If you receive a Task Manager Warning message box warning you of system instability, click the Yes button.) If you have a multiprocessor system, you also need to change the affinity of the process: right-click and select Set Affinity. Then clear all other CPUs except for CPU 0.

8. Run another copy of CPU Stress. In this copy, change the activity level of Thread 1 from Low to Maximum.

9. Now switch back to Performance Monitor. You should see CPU activity every six or so seconds because the thread is boosted to priority 15. You can force updates to occur more frequently than every second by pausing the display with Ctrl+F, and then pressing Ctrl+U, which forces a manual update of the counters. Keep Ctrl+U pressed for continual refreshes.

When you've finished, exit Performance Monitor and the two copies of CPU Stress.

EXPERIMENT: “Listening” to Priority Boosting

To “hear” the effect of priority boosting for CPU starvation, perform the following steps on a system with a sound card:

1. Because of MMCSS’ priority boosts (which we will describe in the next subsection), you need to stop the MultiMedia Class Scheduler Service by opening the Services management interface (Start, Programs, Administrative Tools, Services).
2. Run Windows Media Player (or some other audio-playback program), and begin playing some audio content.
3. Run Cpustres, and set the activity level of Thread 1 to Maximum.
4. Use Task Manager to set the affinities of both Windows Media Player and Cpustres to a single CPU.
5. Raise the priority of Thread 1 of Cpustres from Normal to Time Critical.
6. You should hear the music playback stop as the computer-bound thread begins consuming all available CPU time.
7. Every so often, you should hear bits of sound as the starved thread in the audio playback process gets boosted to 15 and runs enough to send more data to the sound card.
8. Stop Cpustres and Windows Media Player, and start the MMCSS service again.

Applying Boosts

Back in *KiExitDispatcher*, you saw that *KiProcessThreadWaitList* is called to process any threads in the deferred ready list. It is here that the boost information passed by the caller is processed. This is done by looping through each *DeferredReady* thread, unlinking its wait blocks (only Active and Bypassed blocks are unlinked), and then setting two key values in the kernel’s thread control block: *AdjustReason* and *AdjustIncrement*. The reason is one of the two Adjust possibilities seen earlier, and the increment corresponds to the boost value. *KiDeferredReadyThread* is then called, which makes the thread ready for execution, by running two algorithms: the quantum and priority selection algorithm, which you are about to see in two parts, and the processor selection algorithm, which is shown in its respective section later in this topic.

Let’s first look at when the algorithm applies boosts, which happens only in the cases where a thread is not in the real-time priority range.

For an *AdjustUnwait* boost, it will be applied only if the thread is not already experiencing an unusual boost and only if the thread has not disabled boosting by calling *SetThreadPriorityBoost*, which sets the *DisableBoost* flag in the KTHREAD. Another situation that can disable boosting in this case is if the kernel has realized that the thread actually exhausted its quantum (but the clock interrupt did not fire to consume it) and the thread came out of a wait that lasted less than two clock ticks.

If these situations are not currently true, the new priority of the thread will be computed by adding the *AdjustIncrement* to the thread's current base priority. Additionally, if the thread is known to be part of a foreground process (meaning that the memory priority is set to MEMORY_PRIORITY_FOREGROUND, which is configured by Win32k.sys when focus changes), this is where the priority-separation boost (*PsPrioritySeparation*) is applied by adding its value on top of the new priority. This is also known as the Foreground Priority boost, which was explained earlier.

Finally, the kernel checks whether this newly computed priority is higher than the current priority of the thread, and it limits this value to an upper bound of 15 to avoid crossing into the real-time range. It then sets this value as the thread's new current priority. If any foreground separation boost was applied, it sets this value in the *ForegroundBoost* field of the KTHREAD, which results in a *PriorityDecrement* equal to the separation boost.

For *AdjustBoost* boosts, the kernel checks whether the thread's current priority is lower than the *AdjustIncrement* (recall this is the priority of the setting thread) and whether the thread's current priority is below *13*. If so, and priority boosts have not been disabled for the thread, the *AdjustIncrement* priority is used as the new current priority, limited to a maximum of *13*. Meanwhile, the *UnusualBoost* field of the KTHREAD contains the boost value, which results in a *PriorityDecrement* equal to the lock ownership boost.

In all cases where a *PriorityDecrement* is present, the quantum of the thread is also recomputed to be the equivalent of only one clock tick, based on the value of *KiLockQuantumTarget*. This ensures that foreground and unusual boosts will be lost after one clock tick instead of the usual two (or other configured value), as will be shown in the next section. This also happens when an *AdjustBoost* is requested but the thread is running at priority 13 or 14 or with boosts disabled.

After this work is complete, *AdjustReason* is now set to *AdjustNone*.

Removing Boosts

Removing boosts is done in *KiDeferredReadyThread* just as boosts and quantum recomputations are being applied (as shown in the previous section). The algorithm first begins by checking the type of adjustment being done.

For an *AdjustNone* scenario, which means the thread became ready due to perhaps a preemption, the thread's quantum will be recomputed if it already hit its target but the clock interrupt has not yet noticed, as long as the thread was running at a dynamic priority level. Additionally, the thread's priority will be recomputed. For an *AdjustUnwait* or *AdjustBoost* scenario on a non-real-time thread, the kernel checks whether the thread silently exhausted its quantum (just as in the prior section). If it did, or if the thread was running with a base priority of 14 or higher, or if no *PriorityDecrement* is present and the thread has completed a wait that lasted longer than two clock ticks, the quantum of the thread is recomputed, as is its priority.

Priority recomputation happens on non-real-time threads, and it's done by taking the thread's current priority, subtracting its foreground boost, subtracting is unusual boost (the combination of these last two items is the *PriorityDecrement*), and finally subtracting one. Finally, this new priority is bounded with the base priority as the lowest bound, and any existing priority decrement is zeroed out (clearing unusual and foreground boosts). This means that in the case of a lock ownership boost, or any of the unusual boosts explained, the entire boost value is now lost. On the other hand, for a regular *AdjustUnwait* boost, the priority naturally trickles down by one due to the subtraction by one. This lowering eventually stops when the base priority is hit due to the lower bound check.

There is another instance where boosts must be removed, which goes through the *KiRemoveBoostThread* function. This is a special-case boost removal, which occurs due to the lock-ownership boost rule, which specifies that the setting thread must lose its boost when donating its current priority to the waking thread (to avoid a lock convoy). It is also used to undo the boost due to targeted DPC-calls as well as the boost against ERESOURCE lock-starvation boost. The only special detail about this routine is that when computing the new priority, it takes special care to separate the *ForegroundBoost* vs. *UnusualBoost* components of the *PriorityDecrement* in order to maintain any GUI foreground-separation boost that the thread accumulated. This behavior, new to Windows 7, ensures that threads relying on the lock-ownership boost do not behave erratically when running in the foreground, or vice-versa.

Figure 5-20 displays an example of how normal boosts are removed from a thread as it experiences quantum end.

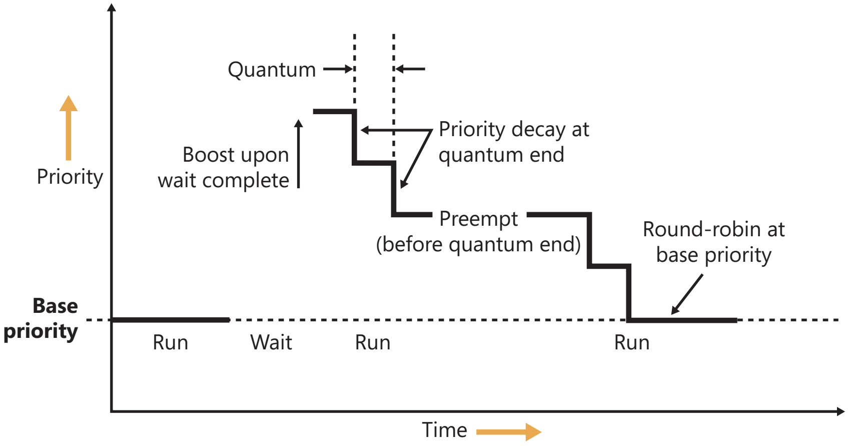

Quantum
Priority
Boost upon wait complete
Priority decay at quantum end
Preempt (before quantum end)
Round-robin at base priority
Base priority
Run
Wait
Run
Run
Time

FIGURE 5-20 Priority boosting and decay

Priority Boosts for Multimedia Applications and Games

As you just saw in the last experiment, although Windows' CPU-starvation priority boosts might be enough to get a thread out of an abnormally long wait state or potential deadlock, they simply cannot deal with the resource requirements imposed by a CPU-intensive application such as Windows Media Player or a 3D computer game.

Skipping and other audio glitches have been a common source of irritation among Windows users in the past, and the user-mode audio stack in Windows makes the situation worse because it offers even more chances for preemption. To address this, client versions of Windows incorporate a service (called MMCSS, described earlier in this chapter) whose purpose is to ensure glitch-free multimedia playback for applications that register with it.

MMCSS works by defining several tasks, including the following:

- Audio
- Capture
- Distribution
- Games
- Playback
- Pro Audio
- Window Manager

Note You can find the settings for MMCSS, including a lists of tasks (which can be modified by OEMs to include other specific tasks as appropriate) in the registry keys under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile. Additionally, the *SystemResponsiveness* value allows you to fine-tune how much CPU usage MMCSS guarantees to low-priority threads.

In turn, each of these tasks includes information about the various properties that differentiate them. The most important one for scheduling is called the Scheduling Category, which is the primary factor determining the priority of threads registered with MMCSS. Table 5-7 shows the various scheduling categories.

TABLE 5-7 Scheduling Categories

Category	Priority	Description
High	23-26	Pro Audio threads running at a higher priority than any other thread on the system except for critical system threads
Medium	16-22	The threads part of a foreground application such as Windows Media Player
Low	8-15	All other threads that are not part of the previous categories
Exhausted	1-7	Threads that have exhausted their share of the CPU and will continue running only if no other higher priority threads are ready to run

The main mechanism behind MMCSS boosts the priority of threads inside a registered process to the priority level matching their scheduling category and relative priority within this category for a guaranteed period of time. It then lowers those threads to the Exhausted category so that other, nonmultimedia threads on the system can also get a chance to execute.

By default, multimedia threads get 80 percent of the CPU time available, while other threads receive 20 percent (based on a sample of 10 ms; in other words, 8 ms and 2 ms, respectively). MMCSS itself runs at priority 27 because it needs to preempt any Pro Audio threads in order to lower their priority to the Exhausted category.

Keep in mind that the kernel still does the actual boosting of the values inside the KTHREAD (MMCSS simply makes the same kind of system call any other application would), and the scheduler is still in control of these threads. It is simply their high priority that makes them run almost uninterrupted on a machine, because they are in the real-time range and well above threads that most user applications run in.

As was discussed earlier, changing the relative thread priorities within a process does not usually make sense, and no tool allows this because only developers understand the importance of the various threads in their programs. On the other hand, because applications must manually register with MMCSS and provide it with information about what kind of thread this is, MMCSS does have the necessary data to change these relative thread priorities (and developers are well aware that this will be happening).

EXPERIMENT: “Listening” to MMCSS Priority Boosting

You’ll now perform the same experiment as the prior one but without disabling the MMCSS service. In addition, you’ll look at the Performance tool to check the priority of the Windows Media Player threads.

1. Run Windows Media Player (because other playback programs might not yet take advantage of the API calls required to register with MMCSS), and begin playing some audio content.
2. If you have a multiprocessor machine, be sure to set the affinity of the Wmplayer.exe process so that it runs on only one CPU (because you’ll use only one CPUSTRES worker thread).
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
4. Click the Add Counter toolbar button (or press Ctrl+N) to bring up the Add Counters dialog box.
5. Select the Thread object, and then select the Priority Current.
6. In the Instances box, type **Wmplayer**, click Search, and then select all its threads. Click the Add button, and then click OK.

7. As in the previous experiment, select Properties from the Action menu. Change the Vertical Scale Maximum to 31 on the Graph tab, set the interval to 1 in Sample Every Seconds of the Graph Elements area on the General tab, and click OK.

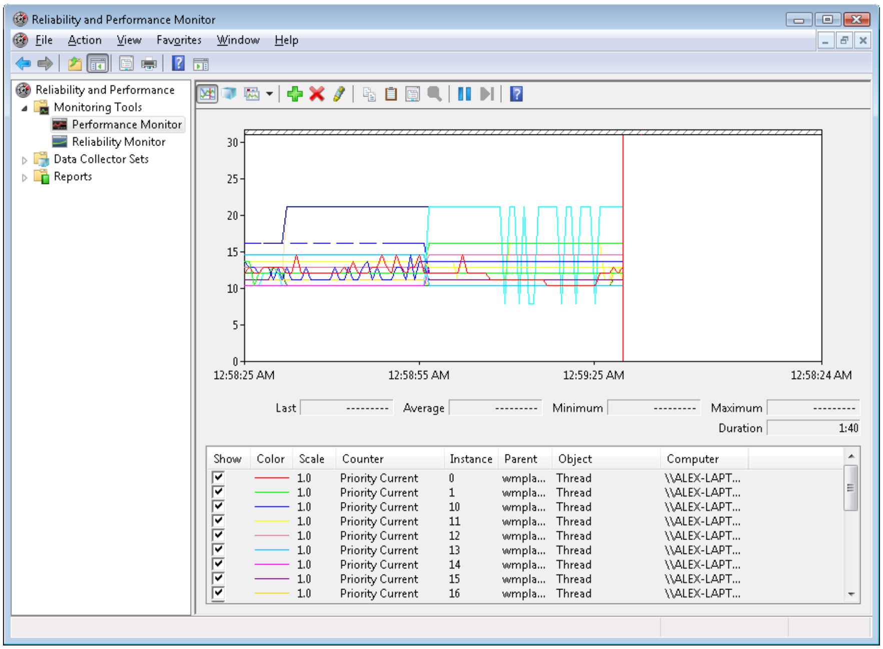
   You should see one or more priority 21 threads inside Wmplayer, which will be constantly running unless there is a higher-priority thread requiring the CPU after they are dropped to the Exhausted category.

8. Run Cpustres, and set the activity level of Thread 1 to Maximum.

9. Raise the priority of Thread 1 from Normal to Time Critical.

10. You should notice the system slowing down considerably, but the music playback will continue. Every so often, you'll be able to get back some responsiveness from the rest of the system. Use this time to stop Cpustres.

11. If the Performance tool was unable to capture data during the time Cpustres ran, run it again, but use Highest instead of Time Critical. This change will slow down the system less, but it still requires boosting from MMCSS. Because once the multimedia thread is put in the Exhausted category there will always be a higher priority thread requesting the CPU (CPUSTRES), you should notice Wmplayer's priority 21 thread drop every so often, as shown here:



Last --------- Average --------- Minimum --------- Maximum ---------
Duration 1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
☑	—	1.0	Priority Current	0	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	1	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	10	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	11	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	12	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	13	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	14	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	15	wmpla...	Thread	\\ALEX-LAPT...
☑	—	1.0	Priority Current	16	wmpla...	Thread	\\ALEX-LAPT...

MMCSS' functionality does not stop at simple priority boosting, however. Because of the nature of network drivers on Windows and the NDIS stack, deferred procedure calls (DPCs) are quite common mechanisms for delaying work after an interrupt has been received from the network card. Because DPCs run at an IRQL level higher than user-mode code (see Chapter 3 for more information on DPCs and IRQLs), long-running network card driver code can still interrupt media playback during network transfers or when playing a game, for example.

Therefore, MMCSS also sends a special command to the network stack, telling it to throttle network packets during the duration of the media playback. This throttling is designed to maximize playback performance, at the cost of some small loss in network throughput (which would not be noticeable for network operations usually performed during playback, such as playing an online game). The exact mechanisms behind it do not belong to any area of the scheduler, so we'll leave them out of this description.

Note The original implementation of the network throttling code had some design issues that caused significant network throughput loss on machines with 1000 Mbit network adapters, especially if multiple adapters were present on the system (a common feature of midrange motherboards). This issue was analyzed by the MMCSS and networking teams at Microsoft and later fixed.

Context Switching

A thread's context and the procedure for context switching vary depending on the processor's architecture. A typical context switch requires saving and reloading the following data:

- Instruction pointer
- Kernel stack pointer
- A pointer to the address space in which the thread runs (the process' page table directory)

The kernel saves this information from the old thread by pushing it onto the current (old thread's) kernel-mode stack, updating the stack pointer, and saving the stack pointer in the old thread's KTHREAD structure. The kernel stack pointer is then set to the new thread's kernel stack, and the new thread's context is loaded. If the new thread is in a different process, it loads the address of its page table directory into a special processor register so that its address space is available. (See the description of address translation in Chapter 10 in Part 2.) If a kernel APC that needs to be delivered is pending, an interrupt at IRQL 1 is requested. (For more information on APCs, see Chapter 3.) Otherwise, control passes to the new thread's restored instruction pointer and the new thread resumes execution.

Scheduling Scenarios

Windows bases the question of "Who gets the CPU?" on thread priority, but how does this approach work in practice? The following sections illustrate just how priority-driven preemptive multitasking works on the thread level.

Voluntary Switch

First a thread might voluntarily relinquish use of the processor by entering a wait state on some object (such as an event, a mutex, a semaphore, an I/O completion port, a process, a thread, a window message, and so on) by calling one of the Windows wait functions (such as *WaitForSingleObject* or *WaitForMultipleObjects*). Waiting for objects is described in more detail in Chapter 3.

Figure 5-21 illustrates a thread entering a wait state and Windows selecting a new thread to run. In Figure 5-21, the top block (thread) is voluntarily relinquishing the processor so that the next thread in the ready queue can run (as represented by the halo it has when in the Running column). Although it might appear from this figure that the relinquishing thread's priority is being reduced, it's not—it's just being moved to the wait queue of the objects the thread is waiting for.

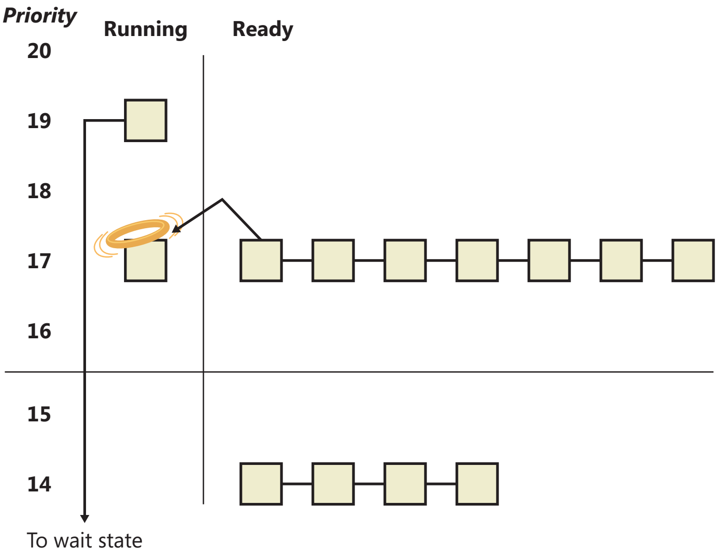


FIGURE 5-21 Voluntary switching

Preemption

In this scheduling scenario, a lower-priority thread is preempted when a higher-priority thread becomes ready to run. This situation might occur for a couple of reasons:

- A higher-priority thread's wait completes. (The event that the other thread was waiting for has occurred.)
- A thread priority is increased or decreased.

In either of these cases, Windows must determine whether the currently running thread should still continue to run or whether it should be preempted to allow a higher-priority thread to run.

Note Threads running in user mode can preempt threads running in kernel mode—the mode in which the thread is running doesn't matter. The thread priority is the determining factor.

When a thread is preempted, it is put at the head of the ready queue for the priority it was running at. Figure 5-22 illustrates this situation.

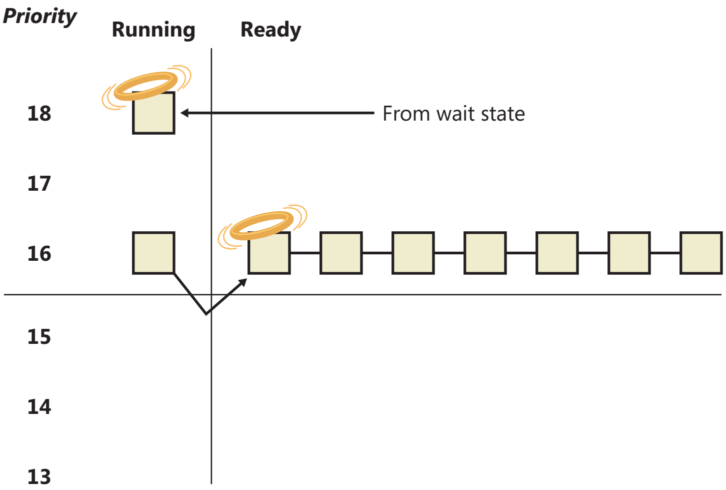


FIGURE 5-22 Preemptive thread scheduling

In Figure 5-22, a thread with priority 18 emerges from a wait state and repossesses the CPU, causing the thread that had been running (at priority 16) to be bumped to the head of the ready queue. Notice that the bumped thread isn't going to the end of the queue but to the beginning; when the preempting thread has finished running, the bumped thread can complete its quantum.

Quantum End

When the running thread exhausts its CPU quantum, Windows must determine whether the thread's priority should be decremented and then whether another thread should be scheduled on the processor.

If the thread priority is reduced, Windows looks for a more appropriate thread to schedule. (For example, a more appropriate thread would be a thread in a ready queue with a higher priority than the new priority for the currently running thread.) If the thread priority isn't reduced and there are other threads in the ready queue at the same priority level, Windows selects the next thread in the ready queue at that same priority level and moves the previously running thread to the tail of that queue (giving it a new quantum value and changing its state from running to ready). This case is illustrated in Figure 5-23. If no other thread of the same priority is ready to run, the thread gets to run for another quantum.

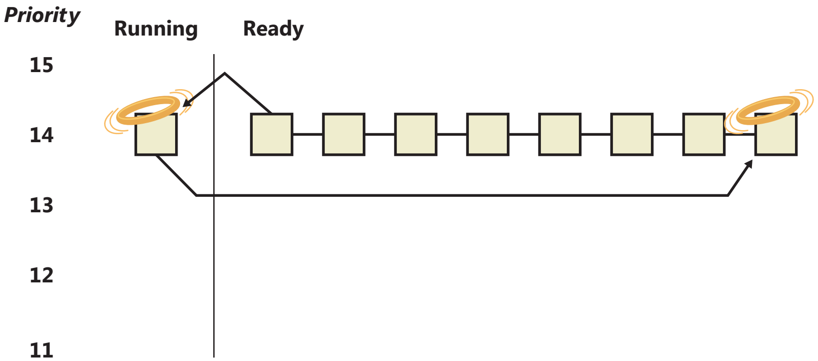


FIGURE 5-23 Quantum end thread scheduling

As you saw, instead of simply relying on a clock interval timer–based quantum to schedule threads, Windows uses an accurate CPU clock cycle count to maintain quantum targets. One factor we haven't yet mentioned is that Windows also uses this count to determine whether quantum end is currently appropriate for the thread—something that might have happened previously and is important to discuss.

Using a scheduling model that relies only on the clock interval timer, the following situation can occur:

- Threads A and B become ready to run during the middle of an interval. (Scheduling code runs not just at each clock interval, so this is often the case.)
- Thread A starts running but is interrupted for a while. The time spent handling the interrupt is charged to the thread.
- Interrupt processing finishes and thread A starts running again, but it quickly hits the next clock interval. The scheduler can assume only that thread A had been running all this time and now switches to thread B.
- Thread B starts running and has a chance to run for a full clock interval (barring pre-emption or interrupt handling).

In this scenario, thread A was unfairly penalized in two different ways. First, the time it spent handling a device interrupt was accounted to its own CPU time, even though the thread probably had nothing to do with the interrupt. (Recall that interrupts are handled in the context of whichever thread was running at the time.) It was also unfairly penalized for the time the system was idling inside that clock interval before it was scheduled.

Figure 5-24 represents this scenario.

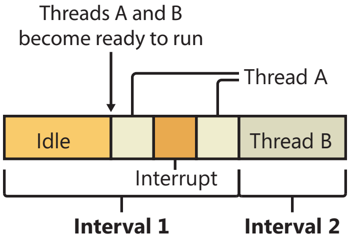


FIGURE 5-24 Unfair time slicing in previous versions of Windows

Because Windows keeps an accurate count of the exact number of CPU clock cycles spent doing work that the thread was scheduled to do (which means excluding interrupts), and because it keeps a quantum target of clock cycles that should have been spent by the thread at the end of its quantum, both of the unfair decisions that would have been made against thread A will not happen in Windows.

Instead, the following situation occurs:

- Threads A and B become ready to run during the middle of an interval.
- Thread A starts running but is interrupted for a while. The CPU clock cycles spent handling the interrupt are not charged to the thread.
- Interrupt processing finishes and thread A starts running again, but it quickly hits the next clock interval. The scheduler looks at the number of CPU clock cycles charged to the thread and compares them to the expected CPU clock cycles that should have been charged at quantum end.
- Because the former number is much smaller than it should be, the scheduler assumes that thread A started running in the middle of a clock interval and might have been additionally interrupted.
- Thread A gets its quantum increased by another clock interval, and the quantum target is recalculated. Thread A now has its chance to run for a full clock interval.
- At the next clock interval, thread A has finished its quantum, and thread B now gets a chance to run.

Figure 5-25 represents this scenario.

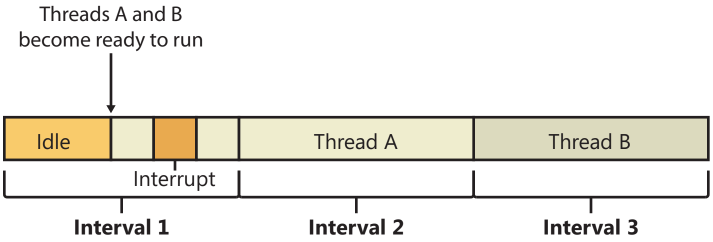


FIGURE 5-25 Fair time slicing in current versions of Windows

Termination

When a thread finishes running (either because it returned from its main routine, called *ExitThread*, or was killed with *TerminateThread*), it moves from the running state to the terminated state. If there are no handles open on the thread object, the thread is removed from the process thread list and the associated data structures are deallocated and released.

Idle Threads

When no runnable thread exists on a CPU, Windows dispatches that CPU's idle thread. Each CPU has its own dedicated idle thread, because on a multiprocessor system one CPU can be executing a thread while other CPUs might have no threads to execute. Each CPU's idle thread is found via a pointer in that CPU's PRCB.

All of the idle threads belong to the idle process. The idle process and idle threads are special cases in many ways. They are, of course, represented by EPROCESS/KPROCESS and ETHREAD/KTHREAD structures, but they are not executive manager processes and thread objects. Nor is the idle process on the system process list. (This is why it does not appear in the output of the kernel debugger's *!process 0 0* command.) However, the idle thread or threads and their process can be found in other ways.

EXPERIMENT: Displaying the Structures of the Idle Threads and Idle Process

The idle thread and process structures can be found in the kernel debugger via the *!pcr* command. "PCR" is short for "processor control region." This command displays a subset of information from the PCR and also from the associated PRCB (processor control block). *!pcr* takes a single numeric argument, which is the number of the CPU whose PCR is to be displayed. The boot processor is processor number 0, and it is always present, so *!pcr 0* should always work. The following output shows the results of this command from a memory dump taken from a 64-bit, four-processor system:

```
3: kd> !pcr 0
KPCR for Processor 0 at fffff800039fdd00:
    Major 1 Minor 1
        NtTib.ExceptionList: fffff80000b95000
            NtTib.StackBase: fffff80000b96080
           NtTib.StackLimit: 000000000008e2d8
         NtTib.SubSystemTib: fffff800039fdd00
              NtTib.Version: 00000000039fde80
          NtTib.UserPointer: fffff800039fe4f0
              NtTib.SelfTib: 000000007efdb000

                    SelfPcr: 0000000000000000
                       Prcb: fffff800039fde80
                       Irql: 0000000000000000
                        IRR: 0000000000000000
                    IDR: 0000000000000000
          InterruptMode: 0000000000000000
                    IDT: 0000000000000000
                    GDT: 0000000000000000
                    TSS: 0000000000000000

          CurrentThread: fffffa8007aa8060
             NextThread: 0000000000000000
             IdleThread: fffff80003a0bcc0

              DpcQueue:
```

This output shows that CPU 0 was executing a thread other than its idle thread at the time the memory dump was obtained, because the *CurrentThread* and *IdleThread* pointers are different. (If you have a multi-CPU system you can try *!pcr 1*, *!pcr 2*, and so on, until you run out; observe that each *IdleThread* pointer is different.)

Now use the *!thread* command on the indicated idle thread address:

```
3: kd> !thread fffff80003a0bcc0
THREAD fffff80003a0bcc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
0000000000000000
  RUNNING on processor 0
Not impersonating
DeviceMap                 fffff8a000008aa0
Owning Process            fffff80003a0c1c0       Image:         Idle
Attached Process          fffffa800792a040       Image:         System
Wait Start TickCount      50774016       Ticks: 12213 (0:00:03:10.828)
Context Switch Count      1147613282
UserTime                  00:00:00.000
KernelTime                8 Days 07:21:56.656
Win32 Start Address nt!KiIdleLoop (0xfffff8000387f910)
Stack Init fffff80000b9cdb0 Current fffff80000b9cd40
Base fffff80000b9d000 Limit fffff80000b97000 Call 0
Priority 16 BasePriority 0 UnusualBoost 0 ForegroundBoost 0 IoPriority 0 PagePriority 0
Child-SP          RetAddr           : Args to Child     [...]: Call Site
fffff800'00b9cd80 00000000'00000000 : fffff800'00b9d000 [...]: nt!KiIdleLoop+0x10d
```

Finally, use the *!process* command on the "Owning Process" shown in the preceding output. For brevity, we'll add a second parameter value of *3*, which causes *!process* to emit only minimal information for each thread:

```
3: kd> !process fffff80003a0c1c0 3
PROCESS fffff80003a0c1c0
    SessionId: none  Cid: 0000    Peb: 00000000  ParentCid: 0000
    DirBase: 00187000  ObjectTable: fffff8a000001630  HandleCount: 1338.
    Image: Idle
    VadRoot fffffa8007846c00 Vads 1 Clone 0 Private 1. Modified 0. Locked 0.
    DeviceMap 0000000000000000
    Token                             fffff8a000004a40
    ElapsedTime                       00:00:00.000
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (6, 50, 450) (24KB, 200KB, 1800KB)
    PeakWorkingSetSize                6
    VirtualSize                       0 Mb
    PeakVirtualSize                   0 Mb
    PageFaultCount                    1
    MemoryPriority                    BACKGROUND
    BasePriority                      0
    CommitCharge                      0

THREAD fffff80003a0bcc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
0000000000000000
  RUNNING on processor 0
THREAD fffff8800310afc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
0000000000000000
  RUNNING on processor 1
THREAD fffff8800317bfc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
0000000000000000
  RUNNING on processor 2
THREAD fffff880031ecfc0  Cid 0000.0000  Teb: 0000000000000000 Win32Thread:
0000000000000000
  RUNNING on processor 3
```

These process and thread addresses can be used with *dt nt!_EPROCESS*, *dt nt!_KTHREAD*, and other such commands as well.

The preceding experiment shows some of the anomalies associated with the idle process and its threads. The debugger indicates an "Image" name of "Idle" (which comes from the EPROCESS structure's *ImageFileName* member), but various Windows utilities report the idle process using different names. Task Manager and Process Explorer call it "System Idle Process," while Tlist calls it "System Process." The process ID and thread IDs (the "client IDs", or "Cid" in the debugger's output) are zero, as are the PEB and TEB pointers, and there are many other fields in the idle process or its threads that might be reported as 0. This occurs because the idle process has no user-mode address space and its threads execute no user-mode code, so they have no need of the various data needed to manage a user-mode environment. Also, the idle process is not an object-manager process object, and its idle threads are not object-manager thread objects. Instead, the initial idle thread and idle process structures are statically allocated and used to bootstrap the system before the process manager and the object manager are initialized. Subsequent idle thread structures are allocated dynamically (as simple allocations from nonpaged pool, bypassing the object manager) as additional processors are brought online. Once process management initializes, it uses the special variable *PsIdleProcess* to refer to the idle process.

Perhaps the most interesting anomaly regarding the idle process is that Windows reports the priority of the idle threads as 0 (16 on x64 systems, as shown earlier). In reality, however, the values of the idle threads' priority members are irrelevant, because these threads are selected for dispatching only when there are no other threads to run. Their priority is never compared with that of any other thread, nor are they used to put an idle thread on a ready queue; idle threads are never part of any ready queues. (Remember, only one thread per Windows system is actually running at priority 0—the zero page thread, explained in Chapter 10 in Part 2.)

Just as the idle threads are special cases in terms of selection for execution, they are also special cases for preemption. The idle thread's routine, *KiIdleLoop*, performs a number of operations that preclude its being preempted by another thread in the usual fashion. When no non-idle threads are available to run on a processor, that processor is marked as idle in its PRCB. After that, if a thread is selected for execution on the idle processor, the thread's address is stored in the *NextThread* pointer of the idle processor's PRCB. The idle thread checks this pointer on each pass through its loop.

Although some details of the flow vary between architectures, the basic sequence of operations of the idle thread is as follows:

1. Enables interrupts briefly, allowing any pending interrupts to be delivered, and then disables them again (using the STI and CLI instructions on x86 and x64 processors). This is desirable because significant parts of the idle thread execute with interrupts disabled.
2. On the debug build on some architectures, checks whether there is a kernel debugger trying to break into the system and, if so, gives it access.
3. Checks whether any DPCs (described in Chapter 3) are pending on the processor. DPCs could be pending if a DPC interrupt was not generated when they were queued. If DPCs are pending, the idle loop calls *KiRetireDpcList* to deliver them. This will also perform timer expiration, as well as deferred ready processing; the latter is explained in the upcoming multiprocessor scheduling section. *KiRetireDpcList* must be entered with interrupts disabled, which is why interrupts are left disabled at the end of step 1. *KiRetireDpcList* exits with interrupts disabled as well.
4. Checks whether a thread has been selected to run next on the processor and, if so, dispatches that thread. This could be the case if, for example, a DPC or timer expiration processed in step 3 resolved the wait of a waiting thread, or if another processor selected a thread for this processor to run while it was already in the idle loop.
5. If requested, checks for threads ready to run on other processors and, if possible, schedules one of them locally. (This operation is explained in the upcoming "Idle Scheduler" section.)
6. Calls the registered power management processor idle routine (in case any power management functions need to be performed), which is either in the processor power driver (such as intelppm.sys) or in the HAL if such a driver is unavailable.

Thread Selection

Whenever a logical processor needs to pick the next thread to run, it calls the *KiSelectNextThread* scheduler function. This can happen in a variety of scenarios:

- A hard affinity change has occurred, making the currently running or standby thread ineligible for execution on its selected logical processor, so another must be chosen.

- The currently running thread reached its quantum end, and the SMT set it was currently running on has now become busy, while other SMT sets within the ideal node are fully idle. (SMT is the abbreviation for Symmetric Multi-Threading, the technical name for the Hyperthreading technology described in Chapter 2.) The scheduler performs a quantum end migration of the current thread, so another must be chosen.
- A wait operation has completed, and there were pending scheduling operations in the wait status register (in other words, the Priority and/or Affinity bits were set).

In these scenarios, the behavior of the scheduler is as follows:

- Call *KiSelectReadyThread* to find the next ready thread that the processor should run, and check whether one was found.
- If a ready thread was not found, the idle scheduler is enabled, and the idle thread is selected for execution.
- Or, if a ready thread was found, it is put in the Standby state and set as the *NextThread* in the KPRCB of the logical processor.

The *KiSelectNextThread* operation is performed only when the logical processor needs to pick, but not yet run, the next schedulable thread (which is why the thread will enter Standby). Other times, however, the logical processor is interested in immediately running the next ready thread or performing another action if one is not available (instead of going idle), such as when the following occurs:

- A priority change has occurred, making the current standby or running thread no longer the highest priority ready thread on its selected logical processor, so a higher priority ready thread must now run.
- The thread has explicitly yielded with *YieldProcessor* or *NtYieldExecution*, and another thread might be ready for execution.
- The quantum of the current thread has expired, and other threads at the same priority level need their chance to run as well
- A thread has lost its priority boost, causing a similar priority change to the scenario just described.
- The idle scheduler is running and needs to check whether a ready thread has not appeared in the interval between which idle scheduling was requested and the idle scheduler ran.

A simple way to remember the difference between which routine runs is to check whether or not the logical processor *must* run a different thread (in which case *KiSelectNextThread* is called) or if it *should, if possible,* run a different thread (in which case *KiSelectReadyThread* is called).

In either case, because each processor has its own database of threads that are ready to run (the dispatcher database's ready queues in the KPRCB), *KiSelectReadyThread* can simply check the current LP's queues, removing the first highest priority thread that it finds, unless this priority is lower than the one of the currently running thread (depending on whether the current thread is still allowed to run, which would not be the case in the *KiSelectNextThread* scenario). If there is no higher priority thread (or no threads are ready at all), no thread is returned.

Idle Scheduler

Whenever the idle thread runs, it checks whether idle scheduling has been enabled, such as in one of the scenarios described in the previous section. If so, the idle thread then begins scanning other processor's ready queues for threads it can run by calling *KiSearchForNewThread*. Note that the runtime costs associated with this operation are not charged as idle thread time, but are instead charged as interrupt and DPC time (charged to the processor), so idle scheduling time is considered system time. The *KiSearchForNewThread* algorithm, which is based on the functions seen in the "Thread Selection" section earlier, will be explained in the upcoming section.

Multiprocessor Systems

On a uniprocessor system, scheduling is relatively simple: the highest-priority thread that wants to run is always running. On a multiprocessor system, it is more complex, because Windows attempts to schedule threads on the most optimal processor for the thread, taking into account the thread's preferred and previous processors, as well as the configuration of the multiprocessor system. Therefore, although Windows attempts to schedule the highest-priority runnable threads on all available CPUs, it guarantees only to be running one of the highest-priority threads somewhere.

Before we describe the specific algorithms used to choose which threads run where and when, let's examine the additional information Windows maintains to track thread and processor state on multiprocessor systems and the three different types of multiprocessor systems supported by Windows (SMT, multicore, and NUMA).

Package Sets and SMT Sets

Windows uses five fields in the KPRCB to determine correct scheduling decisions when dealing with logical processor topologies. The first field, *CoresPerPhysicalProcessor*, determines whether this logical processor is part of a multicore package, and it's computed from the CPUID returned by the processor and rounded to a power of two. The second field, *LogicalProcessorsPerCore* determines whether the logical processor is part of an SMT set, such as on an Intel processor with *HyperThreading* enabled, and is also queried through CPUID and rounded. Multiplying these two numbers yields the number of logical processors per package, or an actual physical processor that fits into a socket. With these numbers, each PRCB can then populate its *PackageProcessorSet* value, which is the affinity mask describing which other logical processors within this group (because packages are constrained to a group) belong to the same physical processor. Similarly, the *CoreProcessorSet* value connects other logical processors to the same core, also called an SMT set. Finally, the *GroupSetMember* value defines which bit mask, within the current processor group, identifies this very logical processor. For example, the logical processor 3 normally has a *GroupSetMember* of 8 (2^3).

EXPERIMENT: Viewing Logical Processor Information

You can examine the information Windows maintains for SMT processors using the *!smt* command in the kernel debugger. The following output is from a dual-core Intel Core i5 system with SMT (four logical processors):

```
SMT Summary:
KeActiveProcessors:
****------------------------------------------------------------ (000000000000000f)
KiIdleSummary:
-*-*------------------------------------------------------------ (000000000000000a)
---------------------------------------------------------------- (0000000000000000)
---------------------------------------------------------------- (0000000000000000)
---------------------------------------------------------------- (0000000000000000)

No PRCB             SMT Set                              APIC Id
  0 fffff8000324ae80 **--------------------------------------------------------------
(0000000000000003) 0x00000000
  1 fffff880009e5180 **--------------------------------------------------------------
(0000000000000003) 0x00000001
  2 fffff88002f65180 --**------------------------------------------------------------
(000000000000000c) 0x00000002
  3 fffff88002fd7180 --**------------------------------------------------------------
(000000000000000c) 0x00000003
Maximum cores per physical processor:   8
Maximum logical processors per core:    2
```

NUMA Systems

Another type of multiprocessor system supported by Windows is one with a nonuniform memory access (NUMA) architecture. In a NUMA system, processors are grouped together in smaller units called nodes. Each node has its own processors and memory and is connected to the larger system through a cache-coherent interconnect bus. These systems are called “nonuniform” because each node has its own local high-speed memory. Although any processor in any node can access all of memory, node-local memory is much faster to access.

The kernel maintains information about each node in a NUMA system in a data structure called KNODE. The kernel variable *KeNodeBlock* is an array of pointers to the KNODE structures for each node. The format of the KNODE structure can be shown using the *dt* command in the kernel debugger, as shown here:

```
lkd> dt nt!_KNODE
   +0x000 PagedPoolSListHead : _SLIST_HEADER
   +0x008 NonPagedPoolSListHead : [3] _SLIST_HEADER
   +0x020 Affinity         : _GROUP_AFFINITY
   +0x02c ProximityId      : Uint4B
   +0x030 NodeNumber       : Uint2B
...
   +0x060 ParkLock         : Int4B
   +0x064 NodePad1         : Uint4B
```

EXPERIMENT: Viewing NUMA Information

You can examine the information Windows maintains for each node in a NUMA system using the *!numa* command in the kernel debugger. The following partial output is from a 64-processor NUMA system from Hewlett-Packard with four processors per node:

```
26: kd> !numa
NUMA Summary:
------------
Number of NUMA nodes : 16
Number of Processors : 64
MmAvailablePages     : 0x03F55E67

KeActiveProcessors   : ****************************************************************
                       (ffffffffffffffff)

NODE 0 (E000000084261900):
    ProcessorMask    : ****------------------------------------------------------------
...
NODE 1 (E0000145FF992200):
    ProcessorMask    : ----****--------------------------------------------------------
...
```

Applications that want to gain the most performance out of NUMA systems can set the affinity mask to restrict a process to the processors in a specific node, although Windows already restricts nearly all threads to a single NUMA node due to its NUMA-aware scheduling algorithms.

How the scheduling algorithms take into account NUMA systems will be covered in the upcoming section "Processor Selection" (and the optimizations in the memory manager to take advantage of node-local memory are covered in Chapter 10 in Part 2).

Processor Group Assignment

While querying the topology of the system to build the various relationships between logical processors, SMT sets, multicore packages and physical sockets, Windows assigns processors to an appropriate group that will describe their affinity (through the extended affinity mask seen earlier). This work is done by the *KePerformGroupConfiguration* routine, which is called during initialization before any other Phase 1 work is done. Note that regardless of the group assignment steps below, NUMA node 0 is always assigned to group 0, no matter what.

First, the function queries all detected nodes (*KeNumberNodes*) and computes the capacity of each node (that is, how many logical processors can be part of the node). This value is stored as the *MaximumProcessors* in the *KeNodeBlock*, which identifies all NUMA nodes on the system. If the system supports NUMA Proximity IDs, the proximity ID is queried for each node as well and saved in the node block. Second, the NUMA distance array is allocated (*KeNodeDistance*), and the distance between each NUMA node is computed as was described in Chapter 3.

The next series of steps deal with specific user-configuration options that override default NUMA assignments. For example, on a system with Hyper-V installed (and the hypervisor configured to auto-start), only one processor group will be enabled, and all NUMA nodes (that can fit) will be associated with group 0. This means that Hyper-V scenarios cannot take advantage of machines with over 64 processors at the moment.

Next, the function checks whether any static group assignment data was passed by the loader (and thus configured by the user). This data specifies the proximity information and group assignment for each NUMA node.

Note Users dealing with large NUMA servers that might need custom control of proximity information and group assignments for testing or validation purposes can input this data through the Group Assignment and Node Distance registry values in the HKLM\SYSTEM \CurrentControlSet\Control\NUMA registry key. The exact format of this data includes a count, followed by an array of proximity IDs and group assignments, which are all 32-bit values.

Before treating this data as valid, the kernel queries the proximity ID to match the node number and then associates group numbers as requested. It then makes sure that NUMA node 0 is associated with group 0, and that the capacity for all NUMA nodes is consistent with the group size. Finally, the function checks how many groups still have remaining capacity.

Next, the kernel dynamically attempts to assign NUMA nodes to groups, while respecting any statically configured nodes if passed-in as we just described. Normally, the kernel tries to minimize the number of groups created, combining as many NUMA nodes as possible per group. However, if this behavior is not desired, it can be configured differently with the /MAXGROUP loader parameter, which is configured through the *maxgroup* BCD option. Turning this value on overrides the default behavior and causes the algorithm to spread as many NUMA nodes as possible into as many groups as possible (while respecting that the currently implemented group limit is 4). If there is only one node, or if all nodes can fit into a single group (and *maxgroup* is off), the system performs the default setting of assigning all nodes to group 0.

If there is more than one node, Windows checks the static NUMA node distances (if any), and then sorts all the nodes by their capacity so that the largest nodes come first. In the group-minimization mode, by adding up all the capacities, the kernel figures out how many maximum processors there can be. By dividing that by the number of processors per group, the kernel assumes there will be this many total groups on the machine (limited to a maximum of 4). In the group-maximization mode, the initial estimate is that there will be as many groups as nodes (limited again to 4).

Now the kernel begins the final assignment process. All fixed assignments from earlier are now committed, and groups are created for those assignments. Next, all the NUMA nodes are reshuffled to minimize the distance between the different nodes within a group. In other words, closer nodes are put in the same group and sorted by distance. Next, the same process is performed for any dynamically configured node to group assignments. Finally, any remaining empty nodes are assigned to group 0.

Logical Processors per Group

Generally, Windows assigns 64 processors per group as explained earlier, but this configuration can also be customized by using different load options, such as the /GROUPSIZE option, which is configured through the *groupsize* BCD element. By specifying a number that is a power of two, groups can be forced to contain fewer processors than normal, for purposes such as testing group awareness in the system (for example, a system with 8 logical processors can be made to appear to have 1, 2, or 4 groups). To force the issue, the /FORCEGROUPAWARE option (BCD element *groupaware*) furthermore makes the kernel avoid group 0 whenever possible, assigning the highest group number available in actions such as thread and DPC affinity selection and process group assignment. Avoid setting a group size of 1, because this will force almost all applications on the system to behave as if they're running on a uniprocessor machine, because the kernel sets the affinity mask of a given process to span only one group until the application requests otherwise (which most applications today will not do).

Note that in the edge case where the number of logical processors in a package cannot fit into a single group, Windows adjusts these numbers so that a package can fit into a single group, shrinking the *CoresPerPhysicalProcessor* number, and if the SMT cannot fit either, doing this as well for *LogicalProcessorsPerCore*. The exception to this rule is if the system actually contains multiple NUMA nodes within a single package. Although this is not a possibility as of this writing, future Multiple-Chip Modules (MCMs, an extension of multicore packages) are due to ship from processor manufacturers in the future. In these modules, two sets of cores as well as two memory controllers are on the same die/package. If the ACPI SRAT table defines the MCM as having two NUMA nodes, depending on group configuration algorithms, Windows might associate the two nodes with two different groups. In this scenario, the MCM package would span more than one group.

Other than causing significant driver and application compatibility problems (which they are designed to identify and root out, when used by developers), these options have an even greater impact on the machine: they will force NUMA behaviors even on a non-NUMA machine. This is because Windows will never allow a NUMA node to span multiple groups, as was shown in the assignment algorithms. So, if the kernel is creating artificially small groups, those two groups must each have their own NUMA node. For example, on a quad-core processor with a group size of two, this will create two groups, and thus two NUMA nodes, which will be subnodes of the main node. This will affect scheduling and memory-management policies in the same way a true NUMA system would, which can be useful for testing.

Logical Processor State

In addition to the ready queues and the ready summary, Windows maintains two bitmasks that track the state of the processors on the system. (How these bitmasks are used is explained in the upcoming section "Processor Selection.") Following are the bitmasks that Windows maintains.

The first one is the active processor mask (*KeActiveProcessors*), which has a bit set for each usable processor on the system. This might be fewer than the number of actual processors if the licensing limits of the version of Windows running supports fewer than the number of available physical processors. To check this, use the variable *KeRegisteredProcessors* to see how many processors are actually licensed on the machine. In this instance, "processors" refers to physical packages. The *KeMaximumProcessors* variable, on the other hand, is the maximum number of logical processors, including all future possible dynamic processor additions, bounded within the licensing limit, and any platform limitations that are queried by calling the HAL and checking with the ACPI SRAT table, if any.

The idle summary (*KiIdleSummary*) is actually an array of two extended bitmasks. In the first entry, called *CpuSet*, each set bit represents an idle processor, while in the second entry, *SMTSet*, each bit describes an idle SMT set.

The nonparked summary (*KiNonParkedSummary*) defines each nonparked logical processor through a bit.

Scheduler Scalability

Because on a multiprocessor system one processor might need to modify another processor's per-CPU scheduling data structures (such as inserting a thread that would like to run on a certain processor), these structures are synchronized by using a per-PRCB queued spinlock, which is held at DISPATCH_LEVEL. Thus, thread selection can occur while locking only an individual processor's PRCB. If needed, up to one more processor's PRCB can also be locked, such as in scenarios of thread stealing, which will be described later. Thread context switching is also synchronized by using a finer-grained per-thread spinlock.

There is also a per-CPU list of threads in the deferred ready state. These represent threads that are ready to run but have not yet been readied for execution; the actual ready operation has been deferred to a more appropriate time. Because each processor manipulates only its own per-processor deferred ready list, this list is not synchronized by the PRCB spinlock. The deferred ready thread list is processed by *KiProcessDeferredReadyList* after a function has already done modifications to process or thread affinity, priority (including due to priority boosting), or quantum values.

This function calls *KiDeferredReadyThread* for each thread on the list, which performs the algorithm shown later in the "Processor Selection" section, which could either cause the thread to run immediately; to be put on the ready list of the processor; or if the processor is unavailable, to be potentially put on a different processor's deferred ready list, in a standby state, or immediately executed. This property is used by the Core Parking engine when parking a core: all threads are put into the deferred ready list, and it is then processed. Because *KiDeferredReadyThread* skips parked cores (as will be shown), it causes all of this processor's threads to wind up on other processors.

Affinity

Each thread has an affinity mask that specifies the processors on which the thread is allowed to run. The thread affinity mask is inherited from the process affinity mask. By default, all processes (and therefore all threads) begin with an affinity mask that is equal to the set of all active processors on their assigned group—in other words, the system is free to schedule all threads on any available processor within the group associated with the process.

However, to optimize throughput, partition workloads to a specific set of processors, or both, applications can choose to change the affinity mask for a thread. This can be done at several levels:

- Calling the *SetThreadAffinityMask* function to set the affinity for an individual thread.
- Calling the *SetProcessAffinityMask* function to set the affinity for all the threads in a process. Task Manager and Process Explorer provide a GUI to this function if you right-click a process and choose Set Affinity. The Psexec tool (from Sysinternals) provides a command-line interface to this function. (See the –a switch in its help output.)
- By making a process a member of a job that has a jobwide affinity mask set using the *SetInformationJobObject* function. (Jobs are described in the upcoming "Job Objects" section.)
- By specifying an affinity mask in the image header when compiling the application. (For more information on the detailed format of Windows images, search for "Portable Executable and Common Object File Format Specification" on *www.microsoft.com*.)

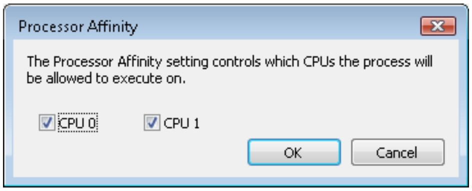
An image can also have the "uniprocessor" flag set at link time. If this flag is set, the system chooses a single processor at process creation time (*MmRotatingProcessorNumber*) and assigns that as the process affinity mask, starting with the first processor and then going round-robin across all the processors within the group. For example, on a dual-processor system, the first time an image marked as uniprocessor is launched, it is assigned to CPU 0; the second time, CPU 1; the third time, CPU 0; the fourth time, CPU 1; and so on. This flag can be useful as a temporary workaround for programs that have multithreaded synchronization bugs that, as a result of race conditions, surface on multiprocessor systems but that don't occur on uniprocessor systems. If an image exhibits such symptoms and is unsigned, the flag can be manually added by editing the image header with a tool such as Imagecfg.exe. A better solution, also compatible with signed executables, is to use the Microsoft Application Compatibility Toolkit and add a shim to force the compatibility database to mark the image as uniprocessor-only at launch time.

EXPERIMENT: Viewing and Changing Process Affinity

In this experiment, you will modify the affinity settings for a process and see that process affinity is inherited by new processes:

1. Run the command prompt (Cmd.exe).
2. Run Task Manager or Process Explorer, and find the Cmd.exe process in the process list.

3. Right-click the process, and select Set Affinity. A list of processors should be displayed. For example, on a dual-processor system you will see this:



The Processor Affinity setting controls which CPUs the process will be allowed to execute on.

☑ CPU 0 ☑ CPU 1

4. Select a subset of the available processors on the system, and click OK. The process' threads are now restricted to run on the processors you just selected.
5. Now run Notepad.exe from the command prompt (by typing **Notepad.exe**).
6. Go back to Task Manager or Process Explorer and find the new Notepad process. Right-click it, and choose Affinity. You should see the same list of processors you chose for the command-prompt process. This is because processes inherit their affinity settings from their parent.

Windows won't move a running thread that could run on a different processor from one CPU to a second processor to permit a thread with an affinity for the first processor to run on the first processor. For example, consider this scenario: CPU 0 is running a priority 8 thread that can run on any processor, and CPU 1 is running a priority 4 thread that can run on any processor. A priority 6 thread that can run on only CPU 0 becomes ready. What happens? Windows won't move the priority 8 thread from CPU 0 to CPU 1 (preempting the priority 4 thread) so that the priority 6 thread can run; the priority 6 thread has to stay in the ready state.

Therefore, changing the affinity mask for a process or a thread can result in threads getting less CPU time than they normally would, because Windows is restricted from running the thread on certain processors. Therefore, setting affinity should be done with extreme care—in most cases, it is optimal to let Windows decide which threads run where.

Extended Affinity Mask

To support more than 64 processors, which is the limit enforced by the affinity mask structure (composed of 64 bits on a 64-bit system), Windows uses an extended affinity mask (KAFFINITY_EX) that is an array of affinity masks, one for each supported processor group (currently defined to 4). When the scheduler needs to refer to a processor in the extended affinity masks, it first de-references the correct bitmask by using its group number and then accesses the resulting affinity directly. In the kernel API, extended affinity masks are not exposed; instead, the caller of the API inputs the group number as a parameter, and receives the legacy affinity mask for that group. In the Windows API, on the other hand, only information about a single group can usually be queried, which is the group of the currently running thread (which is fixed).

The extended affinity mask and its underlying functionality are also how a process can escape the boundaries of its original assigned processor group. By using the extended affinity APIs, threads in a process can choose affinity masks on other processor groups. For example, if a process has 4 threads and the machine has 256 processors, thread 1 can run on processor 4, thread 2 can run on processor 68, thread 3 on processor 132, and thread 4 on processor 196, if each thread set an affinity mask of 0x10 (0b10000 in binary) on groups 0, 1, 2, and 3. Alternatively, the threads can each set an affinity of 0xFFFFFFFF for their given group, and the process then can execute its threads on any available processor on the system (with the limitation, that each thread is restricted to running within its own group only).

Taking advantage of extended affinity must be done at creation time, by specifying a group number in the thread attribute list when creating a new thread. (See the previous topic on thread creation for more information on attribute lists.)

System Affinity Mask

Because Windows drivers usually execute in the context of the calling thread or in the context of an arbitrary thread (that is, not in the safe confines of the System process), currently running driver code might be subject to affinity rules set by the application developer, which are not currently relevant to the driver code and might even prevent correct processing of interrupts and other queued work. Driver developers therefore have a mechanism to temporarily bypass user thread affinity settings, by using the APIs *KeSetSystemAffinityThread*(*Ex*)/*KeSetSystemGroupAffinityThread* and *KeRevertToUserAffinityThread(Ex)/KeRevertToUserGroupAffinityThread*.

Ideal and Last Processor

Each thread has three CPU numbers stored in the kernel thread control block:

- Ideal processor, or the preferred processor that this thread should run on
- Last processor, or the processor on which the thread last ran
- Next processor, or the processor that the thread will be, or is already, running on

The ideal processor for a thread is chosen when a thread is created using a seed in the process control block. The seed is incremented each time a thread is created so that the ideal processor for each new thread in the process rotates through the available processors on the system. For example, the first thread in the first process on the system is assigned an ideal processor of 0. The second thread in that process is assigned an ideal processor of 1. However, the next process in the system has its first thread's ideal processor set to 1, the second to 2, and so on. In that way, the threads within each process are spread across the processors.

Note that this assumes the threads within a process are doing an equal amount of work. This is typically not the case in a multithreaded process, which normally has one or more housekeeping threads and then a number of worker threads. Therefore, a multithreaded application that wants to take full advantage of the platform might find it advantageous to specify the ideal processor numbers for its threads by using the *SetThreadIdealProcessor* function. To take advantage of processor groups, developers should call *SetThreadIdealProcessorEx* instead, which allows selection of a group number for the affinity.

64-bit Windows uses the Stride field in the KPRCB to balance the assignment of newly created threads within a process. The stride is a scalar number that represents the number of affinity bits within a given NUMA node that must be skipped to attain a new independent logical processor slice, where “independent” means on another core (if dealing with an SMT system) or another package (if dealing with a non-SMT but multicore system). Because 32-bit Windows doesn’t support large processor configuration systems, it doesn’t use a stride, and it simply selects the next processor number, trying to avoid sharing the same SMT set if possible. For example, on a dual-processor SMT system with four logical processors, if the ideal processor for the first thread is assigned to logical processor 0, the second thread would be assigned to logical processor 2, the third thread to logical processor 1, the fourth thread to logical process 3, and so forth. In this way, the threads are spread evenly across the physical processors.

Ideal Node

On NUMA systems, when a process is created, an ideal node for the process is selected. The first process is assigned to node 0, the second process to node 1, and so on. Then the ideal processors for the threads in the process are chosen from the process’ ideal node. The ideal processor for the first thread in a process is assigned to the first processor in the node. As additional threads are created in processes with the same ideal node, the next processor is used for the next thread’s ideal processor, and so on.

Thread Selection on Multiprocessor Systems

Before covering multiprocessor systems in more detail, I should summarize the algorithms discussed in the “Thread Selection” section. They either continued executing the current thread (if no new candidate was found) or started running the idle thread (if the current thread had to block). However, there is a third algorithm for thread selection, which was hinted at in the “Idle Scheduler” section earlier, called *KiSearchForNewThread*. This algorithm is called in one specific instance: when the current thread is about to block due to a wait on an object, including when doing an *NtDelayExecutionThread* call, also known as the Sleep API in Windows.

Note This shows a subtle difference between the commonly used Sleep(1) call, which makes the current thread block until the next timer tick, and the *SwitchToThread()* call, which was shown earlier. The “sleep” will use the algorithm about to be described, while the “yield” uses the previously shown logic.

KiSearchForNewThread initially checks whether there is already a thread that was selected for this processor (by reading the *NextThread* field); if so, it dispatches this thread immediately in the Running state. Otherwise, it calls the *KiSelectReadyThread* routine and, if a thread was found, performs the same steps.

If a thread was not found, however, the processor is marked as idle (even though the idle thread is not yet executing) and a scan of other logical processors queues is initiated (unlike the other standard algorithms, which would now give up). Also, because the processor is now considered idle, if the Dynamic Fair Share Scheduling mode (described in the next topic) is enabled, a thread will be released from the idle-only queue if possible and scheduled instead. On the other hand, if the processor core is now parked, the algorithm will not attempt to check other logical processors, as it is preferable to allow the core to enter the parking state instead keeping it busy with new work.

Barring these two scenarios, the work-stealing loop now runs. This code looks at the current NUMA node and removes any idle processors (because they shouldn't have threads that need stealing). Then, starting from the highest numbered processor, the loop calls *KiFindReadyThread* but points it to the remote KPRCB instead of the current one, causing this processor to find the best ready thread from the other processor's queue. If this is unsuccessful and Dynamic Fair Share Scheduler is enabled, a thread from the idle-only queue of the remote logical processor is released on the current processor instead, if possible.

If no candidate ready thread is found, the next lower numbered logical processor is attempted, and so on, until all logical processors have been exhausted on the current NUMA node. In this case, the algorithm keeps searching for the next closest node, and so on, until all nodes in the current group have been exhausted. (Recall that Windows allows a given thread to have affinity only on a single group.) If this process fails to find any candidates, the function returns NULL and the processor enters the idle thread in the case of a wait (which will skip idle scheduling). If this work was already being done from the idle scheduler, the processor enters a sleep state.

Processor Selection

Up until now, we've described how Windows picks a thread when a logical processor needs to make a selection (or when a selection must be made for a given logical processor) and assumed the various scheduling routines have an existing database of ready threads to choose from. Now we'll see how this database gets populated in the first place—in other words, how Windows chooses which LP's ready queues a given ready thread will be associated with. Having described the types of multiprocessor systems supported by Windows as well as the thread affinity and ideal processor settings, we're now ready to examine how this information is used for this purpose.

Choosing a Processor for a Thread When There Are Idle Processors

When a thread becomes ready to run, the *KiDeferredReadyThread* scheduler function is called, causing Windows to perform two tasks: adjust priorities and refresh quantums as needed, as was explained in the "Priority Boosts" section, and then pick the best logical processor for the thread.

Windows first looks up the thread's ideal processor, and then it computes the set of idle processors within the thread's hard affinity mask. This set is then pruned as follows:

- Any idle logical processors that have been parked by the Core Parking mechanism are removed. (See Chapter 9, "Storage Management," in Part 2 for more information on Core Parking.) If this causes no idle processors to remain, idle processor selection is aborted, and the scheduler behaves as if no idle processors were available (which is described in the upcoming section)
- Any idle logical processors that are not on the ideal node (defined as the node containing the ideal processor) are removed, unless this would cause all idle processors to be eliminated.
- On an SMT system, any non-idle SMT sets are removed, even if this might cause the elimination of the ideal processor itself. In other words, Windows prioritizes a non-ideal, idle SMT set over an ideal processor.
- Windows then checks whether the ideal processor is among the remaining set of idle processors. If it isn't, it must then find the most appropriate idle processor. It does so by first checking whether the processor that the thread last ran on is part of the remaining idle set. If so, this processor is considered to be a temporary ideal processor and chosen. (Recall that the ideal processor attempts to maximize processor cache hits, and picking the last processor a thread ran on is a good way of doing so.)
- If the last processor is not part of the remaining idle set, Windows next checks whether the current processor (that is, the processor currently executing this scheduling code) is part of this set; if so, it applies the same logic as in the prior step.
- If neither the last nor the current processor is idle, Windows performs one more pruning operation, by removing any idle logical processors that are not on the same SMT set as the ideal processor. If there are none left, Windows instead removes any processors not on the SMT set of the current processor, unless this, too, eliminates all idle processors. In other words, Windows prefers idle processors that share the same SMT set as the unavailable ideal processor and/or last processor it would've liked to pick in the first place. Because SMT implementations share the cache on the core, this has nearly the same effect as picking the ideal or last processor from the caching perspective.
- Finally, if this last step results in more than one processor remaining in the idle set, Windows picks the lowest numbered processor as the thread's current processor.

Once a processor has been selected for the thread to run on, that thread is put in the standby state and the idle processor's PRCB is updated to point to this thread. If the processor is idle, but not halted, a DPC interrupt is sent so that the processor handles the scheduling operation immediately.

Whenever such a scheduling operation is initiated, *KiCheckForThreadDispatch* is called, which will realize that a new thread has been scheduled on the processor and cause an immediate context switch if possible (as well as pending APC deliveries), or it will cause a DPC interrupt to be sent.

Choosing a Processor for a Thread When There Are No Idle Processors

If there are no idle processors when a thread wants to run, or if the only idle processors were eliminated by the first pruning (which got rid of parked idle processors), Windows first checks whether the latter situation has occurred. In this scenario, the scheduler calls *KiSelectCandidateProcessor* to ask the Core Parking engine for the best candidate processor. The Core Parking engine selects the highest-numbered processor that is unparked within the ideal node. If there are no such processors, the engine forcefully overrides the park state of the ideal processor and causes it to be unparked. Upon returning to the scheduler, it will check whether the candidate it received is idle; if so, it will pick this processor for the thread, following the same last steps as in the previous scenario.

If this fails, Windows compares the priority of the thread running (or the one in the standby state) on the thread's ideal processor to determine whether it should preempt that thread.

If the thread's ideal processor already has a thread selected to run next (waiting in the standby state to be scheduled) and that thread's priority is less than the priority of the thread being readied for execution, the new thread preempts that first thread out of the standby state and becomes the next thread for that CPU. If there is already a thread running on that CPU, Windows checks whether the priority of the currently running thread is less than the thread being readied for execution. If so, the currently running thread is marked to be preempted, and Windows queues a DPC interrupt to the target processor to preempt the currently running thread in favor of this new thread.

If the ready thread cannot be run right away, it is moved into the ready state on the priority queue appropriate to its thread priority, where it will await its turn to run. As seen in the scheduling scenarios earlier, the thread will be inserted either at the head or the tail of the queue, based on whether it entered the ready state due to preemption.

As such, regardless of the underlying scenario and various possibilities, note that threads are always put on their ideal processor's per-processor ready queues, guaranteeing the consistency of the algorithms that determine how a logical processor picks a thread to run.

Processor Share-Based Scheduling

In the previous section, the standard thread-based scheduling implementation of Windows was described, which has served general user and server scenarios reliably since its appearance in the first Windows NT release (with scalability improvements done throughout each release). However, because thread-based scheduling attempts to fairly share the processor or processors only among competing threads of same priority, it does not take into account higher-level requirements such as the distribution of threads to users and the potential for certain users to benefit from more overall CPU time at the expense of other users. This kind of behavior, as it turns out, is highly sought after in terminal-services environments, where dozens of users can be competing for CPU time and a single high-priority thread from a given user has the potential to starve threads from all users on the machine if only thread-based scheduling is used.

Dynamic Fair Share Scheduling

In this section, two alternative scheduling modes implemented by recent versions of Windows will be described: the session-based Dynamic Fair Share Scheduler (DFSS) and an older, legacy SID-based CPU Rate Limit implementation.

DFSS Initialization

During the very last parts of system initialization, as the SOFTWARE hive is initialized by *Smss*, the process manager initiates the final post-boot initialization in *PsBootPhaseComplete*, which calls *PsInitializeCpuQuota*. It is here that the system decides which of the two CPU quota mechanisms (DFSS or legacy) will be employed. For DFSS to be enabled, the *EnableCpuQuota* registry value must be set to 1 in both of the two quota keys: HKLM\SOFTWARE\Policies\Microsoft\Windows\Session Manager\Quota System for the policy-based setting (that can be configured through the Group Policy Editor under Computer Configuration\Administrative Templates\Windows Components \Remote Desktop Services\Remote Desktop Session Host\Connections - Turn off Fair Share CPU Scheduling), as well as under the system key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Quota System, which determines if the system supports the functionality (which, by default, is set to TRUE on Windows Server with the Remote Desktop role).

Note Due to a bug (which you can learn more about at *http://technet.microsoft.com/en-us/library/ee808941(WS.10).aspx*), the group policy setting to turn off DFSS is not honored. The system setting must be manually turned off.

If DFSS is enabled, the *PsCpuFairShareEnabled* variable is set to *true*, which will instruct the kernel, through various scheduling code paths, to behave differently and/or to call into the DFSS engine. Additionally, the default quota is set up to 150 milliseconds for each DFSS cycle, a number called credit that will be explained in more detail shortly.

Once DFSS is enabled, the global *PspCpuQuotaControl* data structure is used to maintain DFSS information, such as the list of per-session CPU quota blocks (as well as a spinlock and count) and the total weight of all sessions on the system. It also stores an array of per-processor DFSS data structures, which you'll see next.

Per-Session CPU Quota Blocks

After DFSS is enabled, whenever a new session is created (other than Session 0), *MiSessionCreate* calls *PsAllocateCpuQuotaBlock* to set up the per-session CPU quota block. The first time this happens on the system (for example, for Session 1), this calls *PspLazyInitializeCpuQuota* to finalize the initialization of DFSS.

This results in the allocation of per-CPU DFSS data structures mentioned in the previous sections, which contain the DPC used for managing the quota (*PspCpuQuotaDpcRoutine*, seen later) and the total number of cycles credited as well as accumulated. This structure also keeps the block generation a monotonically increasing sequence to guarantee atomicity, as well as keeping the idle-only queue lock protecting the list of the same name, which is a central element of the DFSS mechanism yet to be described. Each per-CPU DFSS data structure, in turn, is connected through a sorted doubly-linked list to the various per-session CPU quota blocks that were mentioned at the beginning of this discussion.

When the first-time initialization of DFSS is complete, *PsAllocateCpuQuotaBlock* can continue, first by allocating the actual CPU quota block for this session. This structure maintains overall accounting information on the session, as well as per-CPU tracking—including the cycles remaining and initially allocated, as well as the idle-only queue itself, in a per-CPU quota entry structure.

To begin with, the session ID is stored, and the CPU share weight is set to its default of *5*. You'll see shortly what a weight is, how it can be computed, and its effects on the DFSS engine. Because the quota block has just been created, the initial cycle values are all set to their maximum value for now. Next, this new per-session CPU block must be visible to the system. Therefore, the *PspCpuQuotaControl* data structure is updated with the new total weight of all sessions (by adding this weight), and the quota block is inserted into the block list (sorted by session ID). Finally, *PspCalculateCpuQuotaBlockCycleCredits* enumerates every other session's quota block and captures the new total weight of the system.

Once this is done, the per-session CPU quota block is finalized, and the memory manager sets it in the *CpuQuotaBlock* field of the MM_SESSION_SPACE structure for this session. Likewise, the current EPROCESS (part of this new session's *CpuQuotaBlock* field) is also updated to point to this session's CPU quota block. Now that the process has received a CPU quota block as soon as it became part of the session, future threads created by this process (including the first thread itself) will be allocated with an extra structure after their typical ETHREAD—a per-process CPU Quota APC structure. Additionally, the ETHREAD's *RateApcState* field will be set to *PsRateApcContained*, indicating that this is an embedded Quota APC, as used by the DFSS mechanism (rather than the pool-allocated legacy APC). Finally, the *CpuThrottled* bit is set in the KTHREAD's *ThreadControlFlags*.

At this point, the global quota-control structure contains a pointer to the DFSS per-CPU data structure array, which itself is linked to all the per-session CPU blocks that have been created for each session and associated with the EPROCESS structure of the member processes. In turn, each thread part of such a process has CPU throttling turned on. There is a per-CPU DPC ready to execute, as well as per-thread APCs for each throttled thread.

When the last process in the session loses all its references, *PsDeleteCpuQuotaBlock* is called. It removes the block from the list, refreshes the total weights, and calls *PspCalculateCpuQuotaBlockCycleCredits* to update all other per-session CPU quota blocks.

Charging of Cycles to Throttled Threads

After everything is set up, the entire DFSS mechanism is triggered by the consumption of CPU cycles—something that was already explained in the earlier sections. In other words, not only are consumed cycles used for quantum accounting and providing finer-grained information to thread

APIs, but they also can be “charged” against the thread (and thus against its quota). This operation is done by the *PsChargeProcessCpuCycles* function that is called whenever a thread has completed the accumulation of cycles in its current execution timeline.

The first operation involves accumulating the additional cycles to the per-CPU DFSS data structure for this processor, increasing the *TotalCyclesAccumulated* value. If this accumulation has reached the total credit, the quota DPC is immediately queued. Once the DPC ultimately executes, it calls *PspStartNewFairShareInterval*, which updates the generation, resets the cycles accumulated, and resets the credit to 150 ms. Finally, the idle-only queue is flushed on each processor associated with a given session. (You’ll see what this queue is and what flushing it entails, later.) This part of the algorithm manages the 150-ms interval that controls DFSS.

A second possibility is that the generation of the per-CPU quota entry contained in the current process’ CPU quota block (owned by the session) does not manage the generation of the current per-CPU DFSS data structure. This generation mismatch suggests that a new interval has been reached and no cycle limits have yet been set, so *PspReplenishCycleCredit* is called to do the work. This reads the per-CPU weight and the total weight that were captured earlier in *PspCalculateCpuQuotaBlockCycleCredits*, and it uses them to set the base cycle allowance for the current per-CPU data inside the process’ CPU quota block. To do this, it uses a simple formula: the process receives the equivalent of its cycle credit (150 ms) divided by the total weight of all sessions on the system. Then the amount of cycles it will be permitted to run for (*CyclesRemaining*) is set to the base cycle allowance multiplied by the weight of this particular session. In other words, the process runs for a fairly-divided chunk of time based on the number of other sessions on the system, calculated as a percentage based on its relative weight compared to the overall system weight. When the computation is completed, the generation is set to match.

In all other cases, *PsChargeProcessCpuCycles* merely subtracts the amount of cycles from *CyclesRemaining* and then calls *PsCheckThreadCpuQuota* to see whether these cycles have been exhausted (reaching zero). Note that this function can sometimes also be called directly from the context switch code when control is about to pass to a thread that has CPU throttling enabled.

PsCheckThreadCpuQuota recovers the CPU quota block for this process (that is, for the session), and then further extracts the precise per-CPU information out of it. Once again, it checks whether the generation does not match, which would indicate this is the first charge for this 150-ms credit cycle, and then it calls *PspReplenishCycleCredit*. Next, it checks whether the CPU quota block for the process indicates there are no more cycles remaining. If cycles still remain, the function returns; otherwise, it prepares to suspend the thread’s execution.

Before stopping execution, the function extracts the per-CPU DPC, making sure that it (or the associated per-thread APC) is not already running. If this operation is happening due to the context-switch scenario brought up earlier, the per-thread APC is queued, which will preempt the thread’s execution as soon as the context switch completes. Otherwise, if this is occurring as result of cycle charging (which happens at DISPATCH_LEVEL or higher), the per-CPU DPC is queued instead, which will later queue the per-thread APC. (This forces a near-immediate response to the CPU quota restriction.) In case further cycle accumulation has occurred past the 150-ms cycle credit, the DPC also calls *PspStartNewFairShareInterval*, which was explained earlier.

CPU Throttling and Quota Enforcement

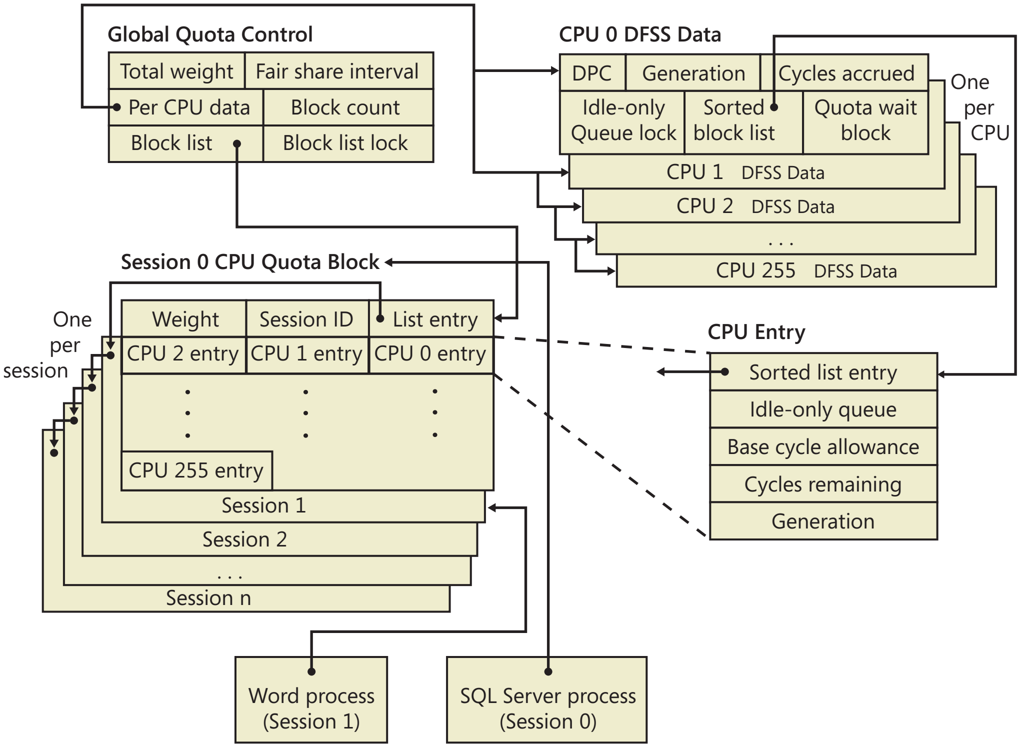
So far, you've seen how DFSS initializes, how CPU quota blocks are created for each session (and then associated with member processes), and how threads running with the CPU throttling bit (implying they are part of processes that are members of a session with DFSS enabled) will consume cycles out of their total weight-relative allowance, resetting every 150 ms. You also saw how, eventually, an APC is queued in all cases where a thread has exhausted its allowed cycles. You'll now see how the APC enforces the CPU quota restriction.

The APC first enters an infinite loop, creating a stack-allocated Quota Wait Block that contains the current thread being restricted, as well as a resume event. It is this event that ultimately allows the thread to continue its execution. Next, the APC gets the per-CPU DFSS data structure pointer and acquires the idle-only queue lock referenced earlier. It then checks whether the idle-only queue on the current processor (which comes from the per-CPU quota entry contained in the process' CPU quota block) is empty. If the list is empty, it implies that this CPU has never been inserted in the sorted block list that is contained in the per-CPU DFSS data structure (part of the *PspCpuQuotaControl* global array). The *PspInsertQuotaBlockCpuEntry* function is thus called to rectify the situation.

Because the DFSS scheduler itself (which has yet to be described) uses this data structure, it must be inserted in the most optimal way—in this case, sorted by the base cycle allowance of each per-CPU data contained within the per-process CPU quota block. Recall that the base cycle allowance is initially the 150-ms credit cycle divided by the total weight of the system (that is, a full allowance), but you'll see how the allowance can be later modified by the DFSS scheduler.

Next, now that the per-CPU Quota Entry is in the sorted block list (or it might already have been if the idle-only queue was not empty), this thread is inserted at the end of the idle-only queue, and it's connected by a linked list entry that's present in the Quota Wait Block. Because this wait block contains the resume event initialized earlier, the DFSS scheduler is able to control the thread when needed.

Finally, the APC enters a wait on this resume event, with the wait reason *WrCpuRateControl*. By using a tool such as Sysinternals PsList, or Process Explorer—all of which display wait reasons (as well as a kernel debugger)—you can see such threads intermittently blocked on a DFSS system.



Resuming Execution

With more and more threads possibly hitting their CPU quota restrictions and block on their respective idle-queues, how will they eventually resume execution? One of the possibilities is that a new 150-ms interval has started. Recall from the earlier discussion that *PspStartNewFairShareInterval* was said to "flush the idle-only queue." This operation, performed by *PspFlushProcessorIdleOnlyQueue*, essentially scans every per-CPU quota entry for this processor (which is located in the sorted block list), and then scans the idle-only queue of each such processor. Picking every thread in the list, the function removes the thread and manually sets the resume event. Thus, any blocked thread on the current CPU gets to resume execution after 150 ms.

Obviously, flushing is not the usual mechanism through which the idle-only queue threads are managed. This work typically is done by the DFSS scheduler itself, which provides the *PsReleaseThreadFromIdleOnlyQueue* routine as a callback that the regular thread scheduler, when the system is about to go idle, can use whenever DFSS-related work is required. Specifically, it is the *KiSearchForNewThread* function, thoroughly described earlier, that calls DFSS in the following two scenarios:

- If *KiSelectReadyThread*, which is called initially, has not found a new thread for the current processor, before it checks other processors' dispatcher ready queues, *KiSearchForNewThread* will ask DFSS to release a thread from the idle-only queue.

- Otherwise, as each CPU's dispatcher ready queues are scanned (by looping *KiSelectReadyThread* calls on each PRCB), if once again no thread is found, the DFSS scheduler is called to release a thread from the idle-only queue on the target processor as well.

Finally, you'll see what work *PsReleaseThreadFromIdleOnlyQueue* actually does and how the DFSS scheduler is implemented.

DFSS Idle-Only Queue Scheduling

PsReleaseThreadFromIdleOnlyQueue initially checks whether the sorted block list is empty (which would imply there aren't even any valid per-CPU quota entries), and it exits if this is the case. Otherwise, it acquires the idle-only queue spinlock from the per-CPU DFSS data structure and calls *PspFindHighestPriorityThreadToRun*. This function scans the sorted block list, recovering every per-CPU quota entry, and then scans every entry (which, if you recall, points to the Quota Wait Block for the thread). Unfortunately, because threads are not inserted by priority (such as real dispatcher ready queues), the entire idle-only queue must be scanned, and the highest priority found to this point is recorded in each iteration. (Because the lock is acquired, no new per-CPU quota entries or idle-only queue threads can be inserted during the scan.)

Note Because DFSS is not truly integrated with the regular thread scheduler, the reason the threads are not sorted by priority is obvious: DFSS is not aware of priority changes after idle-only queue threads have been inserted in its lists. A user could still modify the priority, and because the thread scheduler does not notify DFSS of this, an incorrect thread would be picked.

Additionally, affinity is carefully checked to ensure only correctly affinitized threads are scanned. Although each idle-only queue contains only threads for the current processor, scenario #2 in the preceding section showed how remote processor idle-only queues can also be scanned. DFSS must ensure that the current CPU will run an appropriate remote-CPU, idle-only thread.

Once the highest priority thread has been found on the current per-CPU quota entry, it is removed from the idle-only queue and returned to the caller. Additionally, if this was the last thread on the idle-only queue, the per-CPU entry is removed from the sorted block list. Therefore, note that the other per-CPU quota entries are not checked unless a runnable highest-priority thread was not found on the first per-CPU quota entry (that is, the one with the highest base cycle allowance).

Once the thread is found, *PsReleaseThreadFromIdleOnlyQueue* resumes its execution and once more queues the DPC responsible for eventually launching the per-thread APC from earlier (after making sure the DPC is not already running). Thus, the APC is never directly queued in this case, because this function runs as part of the thread scheduler, already at DISPATCH_LEVEL. Additionally, it wouldn't make sense to queue another per-thread APC just to notify the original APC; instead, the DPC itself will wake up the thread.

This is done by a special check in the DPC routine that checks whether the *ThreadWaitBlockForRelease* field in the per-CPU DFSS data structure is set. If so, the DPC knows that this is a wake-up, not a stop, request, and it sets the resume event associated with the Quota Wait Block. Additionally, it forces the Idle Scheduler on the current CPU to run, by setting the *IdleSchedule* field in the KPRCB that was brought up in the earlier idle scheduler section.

One detail has been glossed over, however: once the idle-only thread is picked, as soon as a context switch is initiated, the cycle accumulation once again detects that the thread has exhausted its cycles, and it re-inserts the thread in the idle-only queue. Therefore, *PsReleaseThreadFromIdleOnlyQueue* must update the cycles remaining for the current per-CPU quota entry, allowing this CPU to run the thread for a little bit longer. How much longer exactly is determined by the value of *KiCyclesPerClockQuantum*, which was shown in the earlier "Quantum" section. Therefore, this CPU is allowed to run the current thread for an entire quantum, at most.

Additionally, the base cycle allowance for this entry must be updated, because the quota for the CPU is actually exhausted and no longer working on a 150-ms cycle credit. Therefore, the allowance is now updated to include an extra *KiCyclesPerClockQuantum* divided by the weight of the session" cycle. Because the base cycle allowance has changed, the sorted block list is reparsed, and the entries are re-sorted correctly to account for this change. Thus, this block will now migrate to the front of the list and have a higher chance to be picked once a future idle-only thread (within this interval) needs to be picked.

Session Weight Configuration

So far, the weight associated to sessions has been described as its default value of 5. However, this weight can be set to anywhere between 1 and 9, and DFSS provides two internal APIs for managing weight information: *PsQueryCpuInformation* and its *Set* equivalent.

Given an array of session handles (to session objects) and associated weights, the Set API sets the new weight for each session, as well as updating the total weight stored in the *PspCpuQuotaControl* global. By calling *PspCalculateCpuQuotaBlockCycleCredits* again, the new settings will be propagated. Likewise, the Query API returns an array of weights and session IDs. The *SeIncreaseQuotaPrivilege* is required in both cases, as well as SESSION_MODIFY_ACCESS for each session whose weight is being modified. Accessing these APIs is done through the native API function *NtQuerySystemInformation*, with the *SystemCpuQuotaInformation* call.

This API, although not provided by the Windows API directly, is what the Windows System Resource Manager uses when the administrator assigns different priorities to different users when the *Weighted_Remote_Sessions* policy is enabled. The three priorities—Premium, Standard, and Basic—map to the 1, 5, and 9 weights in the internal DFSS scheduler mechanism, respectively.

CPU Rate Limits

As part of the hard quota management system in Windows (based on the original soft-limit quota support present since the first version of Windows NT), support for limiting CPU usage exists in the system in three different ways: per-session, per-user, or per-system. Unfortunately, there is no tool that is part of the operating system that allows you to set these limits—you must modify the registry settings manually. Because all the quotas—save one—are memory quotas, we will cover those in Chapter 10 in Part 2, which deals with the memory manager, and instead focus our attention here on the CPU rate limit.

Note See the topic "CPU rate limits in Windows Server 2008 R2 and Windows 7" in the Microsoft Technet Knowledge Articles at *http://technet.microsoft.com/en-us/library/ff384148(WS.10).aspx* for further documentation and examples on when to use CPU rate limits.

The new quota system can be accessed through the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\QuotaSystem, as well as through the standard *NtSetInformationProcess* system call. CPU rate limits can therefore be set in one of three ways:

- By creating a new DWORD value called *CpuRateLimit* and entering the rate information.
- By creating a new key with the security ID (SID) of the account you want to limit, and creating a *CpuRateLimit* DWORD value inside that key.
- By calling *NtSetInformationProcess* and giving it the process handle of the process to limit and the CPU rate limiting information, if the process is tied to the system quota block.

In all three cases, the CPU rate limit data is a straightforward value; it is simply a rate limit expressed as a percentage. For example, to limit a user's applications to consume at most 10% of CPU time, you set *CpuRateLimit* to *10*. The process manager, which is responsible for enforcing the CPU rate limit, uses various system mechanisms to do its job. First, rate limiting works reliably because of the CPU cycle count improvements discussed earlier, which allow the process manager to accurately determine how much CPU time a process has taken and know whether the limit should be enforced. It then uses a combination of DPC and APC routines to throttle down DPC and APC CPU usage, which are outside the direct control of user-mode developers but still result in CPU usage in the system (in the case of a systemwide CPU rate limit).

Finally, the main mechanism through which rate limiting works is by creating an artificial wait on an event object (making the thread uniquely bound to this object and putting it in a wait state, which does not consume CPU cycles). Threads that are artificially waiting because of CPU rate limits can be observed because their wait reason code is set to *WrCpuRateControl*. This mechanism operates through the normal routine of an APC object queued to the thread or threads inside the process currently responsible for the work. The event is eventually signaled by the DPC routine associated with a timer (firing every half a second) responsible for replenishing systemwide CPU usage requests.

Dynamic Processor Addition and Replacement

As you've seen, developers can fine-tune which threads are allowed to (and in the case of the ideal processor, should) run on which processor. This works fine on systems that have a constant number of processors during their run time. (For example, desktop machines require shutting down the computer to make any sort of hardware changes to the processor or their count.)

Today's server systems, however, cannot afford the downtime that CPU replacement or addition normally requires. In fact, one example of when adding a CPU is required for a server is at times of high load that is above what the machine can support at its current level of performance. Having to shut down the server during a period of peak usage would defeat the purpose. To meet this requirement, the latest generation of server motherboards and systems support the addition of processors (as well as their replacement) while the machine is still running. The ACPI BIOS and related hardware on the machine have been specifically built to allow and be aware of this need, but operating system participation is required for full support.

Dynamic processor support is provided through the HAL, which notifies the kernel of a new processor on the system through the function *KeStartDynamicProcessor*. This routine does similar work to that performed when the system detects more than one processor at startup and needs to initialize the structures related to them. When a dynamic processor is added, various system components perform some additional work. For example, the memory manager allocates new pages and memory structures optimized for the CPU. It also initializes a new DPC kernel stack while the kernel initializes the global descriptor table (GDT), the interrupt Dispatch table (IDT), the processor control region (PCR), the process control block (PRCB), and other related structures for the processor.

Other executive parts of the kernel are also called, mostly to initialize the per-processor look-aside lists for the processor that was added. For example, the I/O manager, executive look-aside list code, cache manager, and object manager all use per-processor look-aside lists for their frequently allocated structures.

Finally, the kernel initializes threaded DPC support for the processor and adjusts exported kernel variables to report the new processor. Different memory-manager masks and process seeds based on processor counts are also updated, and processor features need to be updated for the new processor to match the rest of the system (for example, enabling virtualization support on the newly added processor). The initialization sequence completes with the notification to the Windows Hardware Error Architecture (WHEA) component that a new processor is online.

The HAL is also involved in this process. It is called once to start the dynamic processor after the kernel is aware of it, and it is called again after the kernel has finished initialization of the processor. However, these notifications and callbacks only make the kernel aware and respond to processor changes. Although an additional processor increases the throughput of the kernel, it does nothing to help drivers.

To handle drivers, the system has a new default executive callback object, the *ProcessorAdd* callback, that drivers can register with for notifications. Similar to the callbacks that notify drivers of power state or system time changes, this callback allows driver code to, for example, create a new worker thread if desirable so that it can handle more work at the same time.

Once drivers are notified, the final kernel component called is the Plug and Play manager, which adds the processor to the system's device node and rebalances interrupts so that the new processor can handle interrupts that were already registered for other processors. CPU-hungry applications are also able to take advantage of newer processors as well.

However, a sudden change of affinity can have potentially breaking changes for a running application (especially when going from a single-processor to a multiprocessor environment) through the appearance of potential race conditions or simply misdistribution of work (because the process might have calculated the perfect ratios at startup, based on the number of CPUs it was aware of). As a result, applications do not take advantage of a dynamically added processor by default—they must request it.

The Windows APIs *SetProcessAffinityUpdateMode* and *QueryProcessAffinityMode* (which use the undocumented *NtSet/QueryInformationProcess* system call) tell the process manager that these applications should have their affinity updated (by setting the *AffinityUpdateEnable* flag in EPROCESS), or that they do not want to deal with affinity updates (by setting the *AffinityPermanent* flag in EPROCESS). Once an application has told the system that its affinity is permanent, it cannot later change its mind and request affinity updates, so this is a one-time change.

As part of *KeStartDynamicProcessor*, a new step has been added after interrupts are rebalanced, which is to call the process manager to perform affinity updates through *PsUpdateActiveProcessAffinity*. Some Windows core processes and services already have affinity updates enabled, while third-party software will need to be recompiled to take advantage of the new API call. The System process, *Svchost* processes, and *Smss* are all compatible with dynamic processor addition.

Job Objects

A job object is a nameable, securable, shareable kernel object that allows control of one or more processes as a group. A job object's basic function is to allow groups of processes to be managed and manipulated as a unit. A process can be a member of only one job object. By default, its association with the job object can't be broken and all processes created by the process and its descendants are associated with the same job object as well. The job object also records basic accounting information for all processes associated with the job and for all processes that were associated with the job but have since terminated.

Jobs can also be associated with an I/O completion port object, which other threads might be waiting for, with the Windows *GetQueuedCompletionStatus* function. This allows interested parties (typically, the job creator) to monitor for limit violation and events that could affect the job's security (such as a new process being created or a process abnormally exiting).