Foreword by Ben Fathi

Corporate Vice President, Windows Core Development, Microsoft Corporation

Microsoft®

5

FIFTH EDITION

Windows® Internals

Covering Windows Server® 2008 and Windows Vista®

Mark E. Russinovich
and David A. Solomon
with Alex Ionescu

Table of Contents

Chapter 5

Processes, Threads, and Jobs

In this chapter, we'll explain the data structures and algorithms that deal with processes, threads, and jobs in the Windows operating system. The first section focuses on the internal structures that make up a process. The second section outlines the steps involved in creating a process (and its initial thread). The internals of threads and thread scheduling are then described. The chapter concludes with a description of the job object.

Where relevant performance counters or kernel variables exist, they are mentioned. Although this book isn't a Windows programming book, the pertinent process, thread, and job Windows functions are listed so that you can pursue additional information on their use.

Because processes and threads touch so many components in Windows, a number of terms and data structures (such as working sets, objects and handles, system memory heaps, and so on) are referred to in this chapter but are explained in detail elsewhere in the book. To fully understand this chapter, you need to be familiar with the terms and concepts explained in Chapters 1 and 2, such as the difference between a process and a thread, the Windows virtual address space layout, and the difference between user mode and kernel mode.

Process Internals

This section describes the key Windows process data structures. Also listed are key kernel variables, performance counters, and functions and tools that relate to processes.

Data Structures

Each Windows process is represented by an executive process (EPROCESS) block. Besides containing many attributes relating to a process, an EPROCESS block contains and points to a number of other related data structures. For example, each process has one or more threads represented by executive thread (ETHREAD) blocks. (Thread data structures are explained in the section "Thread Internals" later in this chapter.) The EPROCESS block and its related data structures exist in system address space, with the exception of the process environment block (PEB), which exists in the process address space (because it contains information that needs to be accessed by user-mode code).

In addition to the EPROCESS block and the PEB, the Windows subsystem process (Csrss) maintains a parallel structure for each process that is executing a Windows program. Finally, the kernel-mode part of the Windows subsystem (Win32k.sys) will also maintain a per-process data structure that is created the first time a thread calls a Windows USER or GDI function that is implemented in kernel mode.

Figure 5-1 is a simplified diagram of the process and thread data structures. Each data structure shown in the figure is described in detail in this chapter.

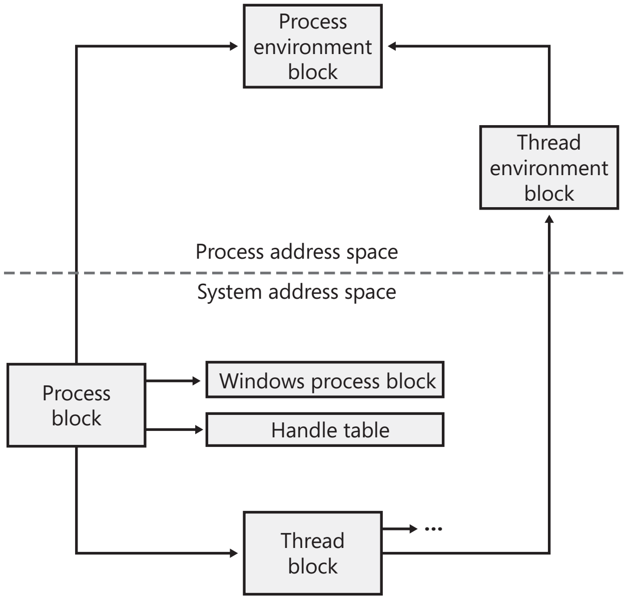


FIGURE 5-1 Data structures associated with processes and threads

First let's focus on the process block. (We'll get to the thread block in the section "Thread Internals" later in the chapter.) Figure 5-2 shows the key fields in an EPROCESS block.

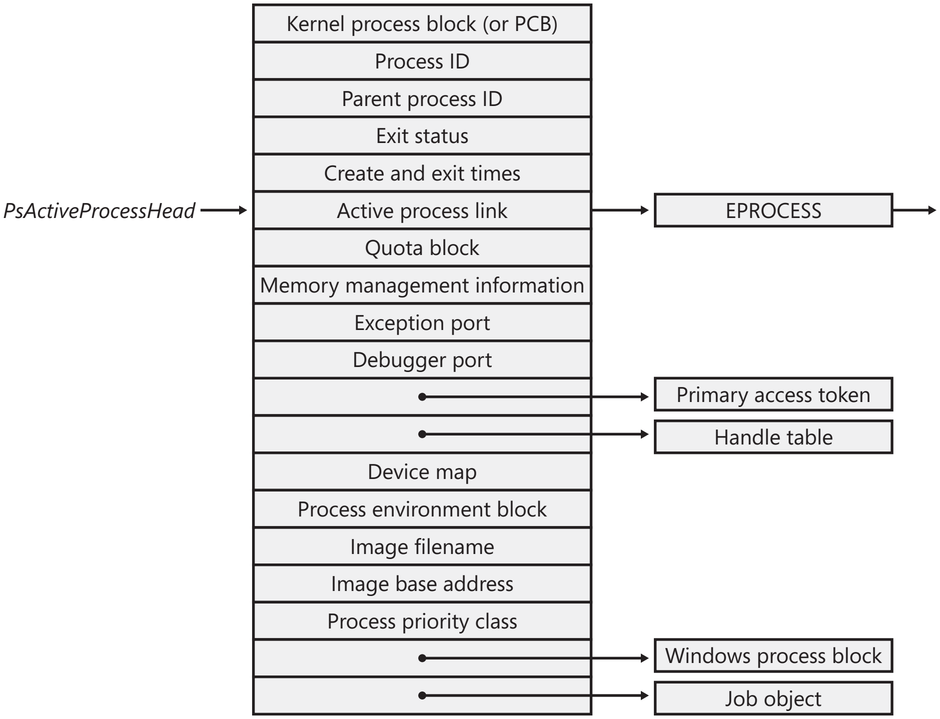


FIGURE 5-2 Structure of an executive process block

EXPERIMENT: Displaying the Format of an EPROCESS Block

For a list of the fields that make up an EPROCESS block and their offsets in hexadecimal, type **dt _eprocess** in the kernel debugger. (See Chapter 1 for more information on the kernel debugger and how to perform kernel debugging on the local system.) The output (truncated for the sake of space) on a 32-bit system looks like this:

```
lkd> dt _eprocess
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x080 ProcessLock      : _EX_PUSH_LOCK
   +0x088 CreateTime       : _LARGE_INTEGER
   +0x090 ExitTime         : _LARGE_INTEGER
   +0x098 RundownProtect   : _EX_RUNDOWN_REF
   +0x09c UniqueProcessId  : Ptr32 Void
   +0x0a0 ActiveProcessLinks : _LIST_ENTRY
   +0x0a8 QuotaUsage       : [3] Uint4B
   +0x0b4 QuotaPeak        : [3] Uint4B
   +0x0c0 CommitCharge     : Uint4B
   +0x0c4 PeakVirtualSize  : Uint4B
   +0x0c8 VirtualSize      : Uint4B
   +0x0cc SessionProcessLinks : _LIST_ENTRY
   +0x0d4 DebugPort        : Ptr32 Void
   +0x0d8 ExceptionPortData : Ptr32 Void
   +0x0d8 ExceptionPortValue : Uint4B
   +0x0d8 ExceptionPortState : Pos 0, 3 Bits
   +0x0dc ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0e0 Token            : _EX_FAST_REF
   +0x0e4 WorkingSetPage   : Uint4B
   +0x0e8 AddressCreationLock : _EX_PUSH_LOCK
   +0x0ec RotateInProgress : Ptr32 _ETHREAD
   +0x0f0 ForkInProgress   : Ptr32 _ETHREAD
   +0x0f4 HardwareTrigger  : Uint4B
   +0x0f8 PhysicalVadRoot  : Ptr32 _MM_AVL_TABLE
   +0x0fc CloneRoot        : Ptr32 Void
   +0x100 NumberOfPrivatePages : Uint4B
   +0x104 NumberOfLockedPages : Uint4B
   +0x108 Win32Process     : Ptr32 Void
   +0x10c Job              : Ptr32 _EJOB
   +0x110 SectionObject    : Ptr32 Void
   +0x114 SectionBaseAddress : Ptr32 Void
   +0x118 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
```

Note that the first field (Pcb) is actually a substructure, the kernel process block (KPROCESS), which is where scheduling-related information is stored. To display the format of the kernel process block, type **dt _kprocess**:

```
lkd> dt _kprocess
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : Uint4B
   +0x01c Unused0          : Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   : _LIST_ENTRY
   +0x058 ProcessLock      : Uint4B
   +0x05c Affinity         : Uint4B
   +0x060 AutoAlignment    : Pos 0, 1 Bit
   +0x060 DisableBoost     : Pos 1, 1 Bit
   +0x060 DisableQuantum   : Pos 2, 1 Bit
   +0x060 ReservedFlags    : Pos 3, 29 Bits
   +0x060 ProcessFlags     : Int4B
   +0x064 BasePriority     : Char
   +0x065 QuantumReset     : Char
   +0x066 State            : UChar
   +0x067 ThreadSeed       : UChar
   +0x068 PowerState       : UChar
   +0x069 IdealNode        : UChar
   +0x06a Visited          : UChar
   +0x06b Flags            : _KEXECUTE_OPTIONS
   +0x06b ExecuteOptions   : UChar
   +0x06c StackCount       : Uint4B
   +0x070 ProcessListEntry : _LIST_ENTRY
   +0x078 CycleTime        : Uint8B
```

An alternative way to see the KPROCESS (and other substructures in the EPROCESS) is to use the recursion (–*r*) switch of the *dt* command. For example, typing **dt _eprocess –r1** will recurse and display all substructures one level deep.

The *dt* command shows the format of a process block, not its contents. To show an instance of an actual process, you can specify the address of an EPROCESS structure as an argument to the *dt* command. You can get the address of all the EPROCESS blocks in the system by using the *!process 0 0* command. An annotated example of the output from this command is included later in this chapter.

Table 5-1 explains some of the fields in the preceding experiment in more detail and includes references to other places in the book where you can find more information about them. As we've said before and will no doubt say again, processes and threads are such integral parts of Windows that it's impossible to talk about them without referring to many other parts of the system. To keep the length of this chapter manageable, however, we've covered those related subjects (such as memory management, security, objects, and handles) elsewhere.

TABLE 5-1 Contents of the EPROCESS Block

Element	Purpose	Additional Reference
Kernel process (KPROCESS) block	Common dispatcher object header, pointer to the process page directory, list of kernel thread (KTHREAD) blocks belonging to the process, default base priority, affinity mask, and total kernel and user time and CPU clock cycles for the threads in the process.	Thread scheduling (Chapter 5)
Process identification	Unique process ID, creating process ID, name of image being run, window station process is running on.	
Quota block	Limits on processor usage, nonpaged pool, paged pool, and page file usage plus current and peak process nonpaged and paged pool usage. (Note: Several processes can share this structure: all the system processes in session 0 point to a single systemwide quota block; all other processes in interactive sessions share a single quota block.)	

Element	Purpose	Additional Reference
Virtual address descriptors (VADs)	Series of data structures that describes the status of the portions of the address space that exist in the process.	Virtual address descriptors (Chapter 9)
Working set information	Pointer to working set list (MMWSL structure); current, peak, minimum, and maximum working set size; last trim time; page fault count; memory priority; outswap flags; page fault history.	Working sets (Chapter 9)
Virtual memory information	Current and peak virtual size, page file usage, hardware page table entry for process page directory.	Chapter 9
Exception legacy local procedure call (LPC) port	Interprocess communication channel to which the process manager sends a message when one of the process's threads causes an exception.	Exception dispatching (Chapter 3)
Debugging object	Executive object through which the user-mode debugging infrastructure sends notifications when one of the process's threads causes a debug event.	User-mode debugging (Chapter 3)
Access token (TOKEN)	Executive object describing the security profile of this process.	Chapter 6
Handle table	Address of per-process handle table.	Object handles and the process handle table (Chapter 3)
Device map	Address of object directory to resolve device name references in (supports multiple users).	Object names (Chapter 3)
Process environment block (PEB)	Image information (base address, version numbers, module list), process heap information, and thread-local storage utilization. (Note: The pointers to the process heaps start at the first byte after the PEB.)	Chapter 5
Windows subsystem process block (W32PROCESS)	Process details needed by the kernel-mode component of the Windows subsystem.	

The kernel process (KPROCESS) block, which is part of the EPROCESS block, and the process environment block (PEB), which is pointed to by the EPROCESS block, contain additional details about the process object. The KPROCESS block (which is sometimes called the PCB or process control block) is illustrated in Figure 5-3. It contains the basic information that the Windows kernel needs to schedule the threads inside a process. (Page directories are covered in Chapter 9, and kernel thread blocks are described in more detail later in this chapter.)

The PEB, which lives in the user process address space, contains information needed by the image loader, the heap manager, and other Windows system DLLs that need to access it from user mode. (The EPROCESS and KPROCESS blocks are accessible only from kernel mode.) The basic structure of the PEB is illustrated in Figure 5-4 and is explained in more detail later in this chapter.

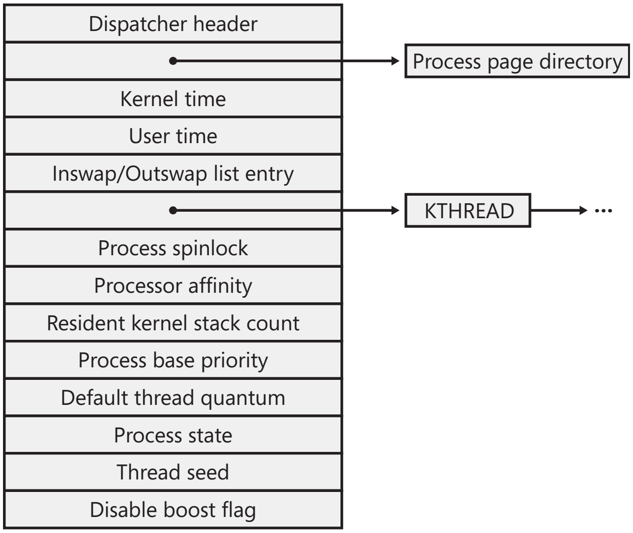


FIGURE 5-3 Structure of the executive process block

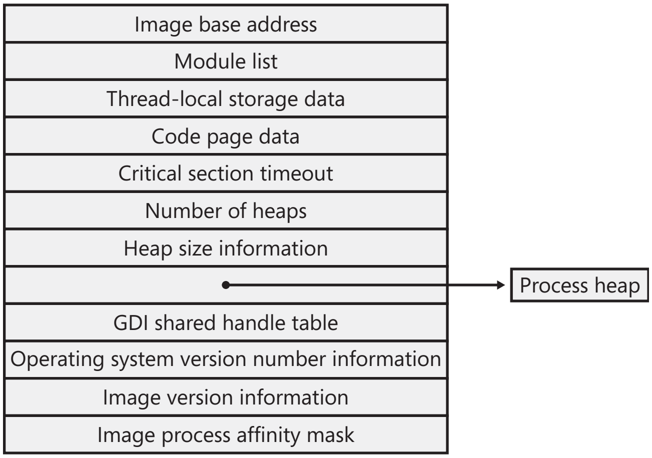


FIGURE 5-4 Fields of the process environment block

EXPERIMENT: Examining the PEB

You can dump the PEB structure with the *!peb* command in the kernel debugger. To get the address of the PEB, use the *!process* command as follows:

```
lkd> !process
PROCESS 8575f030  SessionId: 1  Cid: 08d0    Peb: 7ffd9000  ParentCid: 0360
    DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  66.
    Image: windbg.exe
```

Then specify that address to the *!peb* command as follows:

```
lkd> !peb 7ffd9000
PEB at 7ffd9000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         002a0000
    Ldr                       77895d00
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00151c38 . 00191558
    Ldr.InLoadOrderModuleList:           00151bb8 . 00191548
    Ldr.InMemoryOrderModuleList:         00151bc0 . 00191550
            Base TimeStamp                     Module
          2a0000 4678a41e Jun 19 23:50:54 2007 C:\Program Files\Debugging Tools for
              Windows\windbg.exe
        777d0000 4549bdc9 Nov 02 05:43:37 2006 C:\Windows\system32\Ntdll.dll
        764c0000 4549bd80 Nov 02 05:42:24 2006 C:\Windows\system32\kernel32.dll
    SubSystemData:     00000000
    ProcessHeap:       00150000
    ProcessParameters: 001512e0
    WindowTitle:  'C:\Users\Alex Ionescu\Desktop\WinDbg.lnk'
    ImageFile:    'C:\Program Files\Debugging Tools for Windows\windbg.exe'
    CommandLine:  '"C:\Program Files\Debugging Tools for Windows\windbg.exe" '
    DllPath:      'C:\Program Files\Debugging Tools for Windows;C:\Windows\
        system32;C:\Windows\system;C:\Windows;.;C:\Windows\system32;C:\Windows;
        C:\Windows\System32\Wbem;C:\Program Files\Common Files\Roxio Shared\
        DLLShared\;C:\Program Files\Common Files\Roxio Shared\DLLShared\;C:\Program
        Files\Common Files\Roxio Shared\9.0\DLLShared\;c:\sysint;C:\Program Files\
        QuickTime\QTSystem\'
    Environment:  001850a8
        ALLUSERSPROFILE=C:\ProgramData
        APPDATA=C:\Users\Alex Ionescu\AppData\Roaming
    .
    .
    .
```

Kernel Variables

A few key kernel global variables that relate to processes are listed in Table 5-2. These variables are referred to later in the chapter, when the steps in creating a process are described.

TABLE 5-2 Process-Related Kernel Variables

Element	Purpose	Additional Reference
PsActiveProcessHead	Doubly linked list	List head of process blocks
PsIdleProcess	Pointer to EPROCESS	Idle process block
PsInitialSystemProcess	Pointer to EPROCESS	Pointer to the process block of the initial system process that contains the system threads
PspCreateProcessNotifyRoutine	Array of executive callback objects	Array of callback objects describing the routines to be called on process creation and deletion (maximum of eight)
PspCreateProcessNotifyRoutineCount	32-bit integer	Count of registered process notification routines
PspCreateProcessNotifyRoutineCountEx	32-bit integer	Count of registered extended process notification routines
PspLoadImageNotifyRoutine	Array of executive callback objects	Array of callback objects describing the routines to be called on image load (maximum of eight)
PspLoadImageNotifyRoutineCount	32-bit integer	Count of registered image-load notification routines
PspNotifyEnableMask	32-bit integer	Mask for quickly checking whether any extended or standard notification routines are enabled
PspCidTable	Pointer to HANDLE_ TABLE	Handle table for process and thread client IDs

Performance Counters

Windows maintains a number of counters with which you can track the processes running on your system; you can retrieve these counters programmatically or view them with the Performance tool. Table 5-3 lists the performance counters relevant to processes.

TABLE 5-3 Process-Related Performance Counters

Object: Counter	Function
Process: % Privileged Time	Describes the percentage of time that the threads in the process have run in kernel mode during a specified interval.
Process: % Processor Time	Describes the percentage of CPU time that the threads in the process have used during a specified interval. This count is the sum of % Privileged Time and % User Time.

Object: Counter	Function
Process: % User Time	Describes the percentage of time that the threads in the process have run in user mode during a specified interval.
Process: Elapsed Time	Describes the total elapsed time in seconds since this process was created.
Process: ID Process	Returns the process ID. This ID applies only while the process exists because process IDs are reused.
Process: Creating Process ID	Returns the process ID of the creating process. This value isn't updated if the creating process exits.
Process: Thread Count	Returns the number of threads in the process.
Process: Handle Count	Returns the number of handles open in the process.

Relevant Functions

For reference purposes, some of the Windows functions that apply to processes are described in Table 5-4. For further information, consult the Windows API documentation in the MSDN Library.

TABLE 5-4 Process-Related Functions

Function	Description
CreateProcess	Creates a new process and thread using the caller's security identification
CreateProcessAsUser	Creates a new process and thread with the specified alternate security token
CreateProcessWithLogonW	Creates a new process and thread to run under the credentials of the specified username and password
CreateProcessWithTokenW	Creates a new process and thread with the specified alternate security token, with additional options such as allowing the user profile to be loaded
OpenProcess	Returns a handle to the specified process object
ExitProcess	Ends a process, and notifies all attached DLLs
TerminateProcess	Ends a process without notifying the DLLs
FlushInstructionCache	Empties the specified process's instruction cache
FlushProcessWriteBuffers	Empties the specified process's write queue
GetProcessTimes	Obtains a process's timing information, describing how much time the threads inside the process spent in user and kernel mode
QueryProcessCycleTimeCounter	Obtains a process's CPU timing information, describing how many clock cycles the threads inside the process have spent in total
Query/ SetProcessAffinityUpdateMode	Defines whether the process's affinity is automatically updated if new processors are added to the running system

Function	Description
Get/SetProcessDEPPolicy	Returns or sets the DEP (Data Execution Protection) policy for the process
GetExitCodeProcess	Returns the exit code for a process, indicating how and why the process shut down
GetCommandLine	Returns a pointer to the command-line string passed to the current process
QueryFullProcessImageName	Returns the full name of the executable image associated with the process
GetCurrentProcess	Returns a pseudo handle for the current process
GetCurrentProcessId	Returns the ID of the current process
GetProcessVersion	Returns the major and minor versions of the Windows version on which the specified process expects to run
GetStartupInfo	Returns the contents of the STARTUPINFO structure specified during *CreateProcess*
GetEnvironmentStrings	Returns the address of the environment block
Get/SetEnvironmentVariable	Returns or sets a specific environment variable
Get/SetProcessShutdownParameters	Defines the shutdown priority and number of retries for the current process
SetProcessDPIAware	Specifies whether the process is aware of dots per inch (DPI) settings
GetGuiResources	Returns a count of User and GDI handles

EXPERIMENT: Using the Kernel Debugger *!process* Command

The kernel debugger *!process* command displays a subset of the information in an EPROCESS block. This output is arranged in two parts for each process. First you see the information about the process, as shown here (when you don't specify a process address or ID, *!process* lists information for the active process on the current CPU):

```
lkd> !process
PROCESS 85857160  SessionId: 1  Cid: 0bcc    Peb: 7ffd9000  ParentCid: 090c
    DirBase: b45b0820  ObjectTable: b94ffda0  HandleCount:  99.
    Image: windbg.exe
    VadRoot 85a1c8e8 Vads 97 Clone 0 Private 5919. Modified 153. Locked 1.
    DeviceMap 9d32ee50
    Token                             ebaa1938
    ElapsedTime                       00:48:44.125
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         166784
    QuotaPoolUsage[NonPagedPool]      4776
    Working Set Sizes (now,min,max)  (8938, 50, 345) (35752KB, 200KB, 1380KB)
    PeakWorkingSetSize                8938
    VirtualSize                       106 Mb
    PeakVirtualSize                   108 Mb
    PageFaultCount                    37066
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      6242
```

After the basic process output comes a list of the threads in the process. That output is explained in the "Experiment: Using the Kernel Debugger *!thread* Command" section later in the chapter. Other commands that display process information include *!handle*, which dumps the process handle table (which is described in more detail in the section "Object Handles and the Process Handle Table" in Chapter 3). Process and thread security structures are described in Chapter 6.

Protected Processes

In the Windows security model, any process running with a token containing the debug privilege (such as an administrator's account) can request any access right that it desires to any other process running on the machine—for example, it can read and write arbitrary process memory, inject code, suspend and resume threads, and query information on other processes. Tools like Process Explorer and Task Manager need and request these access rights to provide their functionality to users.

This logical behavior (which helps ensure that administrators will always have full control of the running code on the system) clashes with the system behavior for digital rights management requirements imposed by the media industry on computer operating systems that need to support playback of advanced, high-quality digital content such as BluRay and HD-DVD media. To support reliable and protected playback of such content, Windows uses *protected processes.* These processes exist alongside normal Windows processes, but they add significant constraints to the access rights that other processes on the system (even when running with administrative privileges) can request.

Protected processes can be created by any application; however, the operating system will only allow a process to be protected if the image file has been digitally signed with a special Windows Media Certificate. The Protected Media Path (PMP) in Windows Vista makes use of protected processes to provide protection for high-value media, and developers of applications such as DVD players can make use of protected processes by using the Media Foundation API.

The Audio Device Graph process (Audiodg.exe) is a protected process, since protected music content may be decoded through it. Similarly, the Windows Error Reporting (WER;

see Chapter 3 for more information) client process (Werfault.exe) can also run protected because it needs to have access to protected processes in case one of them crashes. Finally, the System process itself is protected because some of the decryption information is generated by the Ksecdd.sys driver and stored in its user-mode memory. The System process is also protected to protect the integrity of all kernel handles (since the System process's handle table contains all the kernel handles on the system).

At the kernel level, support for protected processes is twofold: first, the bulk of process creation occurs in kernel mode to avoid injection attacks. (The flow for both protected and standard process creation is described in detail in the next section.) Second, protected processes have a special bit set in their EPROCESS structure that modifies the behavior of security-related routines in the process manager to deny certain access rights that would normally be granted to administrators. Table 5-5 indicates access rights that are limited or denied.

TABLE 5-5 Process Access Rights Denied for Protected Processes

Object: Access Mask	Function
Standard: READ_CONTROL	Prevents the protected process's access control list (ACL) from being read.
Standard: WRITE_DAC, WRITE_OWNER	Prevents access to the protected process's access control list or modifying its owner (which would grant the former).
Process: PROCESS_ALL_ACCESS	Prevents full access to the protected process.
Process: PROCESS_CREATE_PROCESS	Prevents creation of a child process of a protected process.
Process: PROCESS_CREATE_THREAD	Prevents creation of a thread inside a protected process.
Process: PROCESS_DUP_HANDLE	Prevents duplication of a handle owned by the protected process.
Process: PROCESS_QUERY_INFORMATION	Prevents querying all information on a protected process. However, a new access right was added, PROCESS_QUERY_LIMITED_INFORMATION, that grants limited access to information on the process.
Process: PROCESS_SET_QUOTA	Prevents setting memory or processor-usage limits on a protected process.
Process: PROCESS_SET_INFORMATION	Prevents modification of process settings for a protected process.
Process: PROCESS_VM_OPERATION, PROCESS_VM_READ, PROCESS_VM_WRITE	Prevents accessing the memory of a protected process.

Certain access rights are also disabled for threads running inside protected processes; we will look at those access rights later in this chapter in the section "Thread Internals."

Because Process Explorer uses standard user-mode Windows APIs to query information on process internals, it is unable to perform certain operations on such processes. On the other hand, a tool like WinDbg in kernel debugging mode, which uses kernel-mode infrastructure to obtain this information, will be able to display complete information. See the experiment in the thread internals section on how Process Explorer behaves when confronted with a protected process such as Audiodg.exe.

Note As mentioned in Chapter 1, to perform local kernel debugging you must boot in debugging mode (enabled by using "bcdedit /debug on" or by using the Msconfig advanced boot options). This protects against debugger-based attacks on protected processes and the Protected Media Path (PMP). When booted in debugging mode, high-definition content playback will not work; for example, attempting to play MPEG2 media such as a DVD will result in an access violation inside the media player (this is by design).

Limiting these access rights reliably allows the kernel to sandbox a protected process from user-mode access. On the other hand, because a protected process is indicated by a flag in the EPROCESS block, an administrator can still load a kernel-mode driver that disables this bit. However, this would be a violation of the PMP model and considered malicious, and such a driver would likely eventually be blocked from loading on a 64-bit system because the kernel-mode code-signing policy prohibits the digital signing of malicious code. Even on 32-bit systems, the driver has to be recognized by PMP policy or else the playback will be halted. This policy is implemented by Microsoft and not by any kernel detection. This block would require manual action from Microsoft to identify the signature as malicious and update the kernel.

Flow of *CreateProcess*

So far in this chapter, you've seen the structures that are part of a process and the API functions with which you (and the operating system) can manipulate processes. You've also found out how you can use tools to view how processes interact with your system. But how did those processes come into being, and how do they exit once they've fulfilled their purpose? In the following sections, you'll discover how a Windows process comes to life.

A Windows subsystem process is created when an application calls one of the process creation functions, such as *CreateProcess*, *CreateProcessAsUser*, *CreateProcessWithTokenW*, or *CreateProcessWithLogonW*. Creating a Windows process consists of several stages carried out in three parts of the operating system: the Windows client-side library Kernel32.dll (in the case of the *CreateProcessAsUser*, *CreateProcessWithTokenW*, and *CreateProcessWithLogonW* routines, part of the work is first done in Advapi32.dll), the Windows executive, and the Windows subsystem process (Csrss).

Because of the multiple environment subsystem architecture of Windows, creating an executive process object (which other subsystems can use) is separated from the work involved in creating a Windows subsystem process. So, although the following description of the flow of the Windows *CreateProcess* function is complicated, keep in mind that part of the work is specific to the semantics added by the Windows subsystem as opposed to the core work needed to create an executive process object.

The following list summarizes the main stages of creating a process with the Windows *CreateProcess* function. The operations performed in each stage are described in detail in the subsequent sections. Some of these operations may be performed by *CreateProcess* itself (or other helper routines in user mode), while others will be performed by *NtCreateUserProcess* or one of its helper routines in kernel mode. In our detailed analysis to follow, we will differentiate between the two at each step required.

> **Note** Many steps of *CreateProcess* are related to the setup of the process virtual address space and therefore refer to many memory management terms and structures that are defined in Chapter 9.

1. Validate parameters; convert Windows subsystem flags and options to their native counterparts; parse, validate, and convert the attribute list to its native counterpart.
2. Open the image file (.exe) to be executed inside the process.
3. Create the Windows executive process object.
4. Create the initial thread (stack, context, and Windows executive thread object).
5. Perform post-creation, Windows-subsystem-specific process initialization.
6. Start execution of the initial thread (unless the CREATE_ SUSPENDED flag was specified).
7. In the context of the new process and thread, complete the initialization of the address space (such as load required DLLs) and begin execution of the program.

Figure 5-5 shows an overview of the stages Windows follows to create a process.

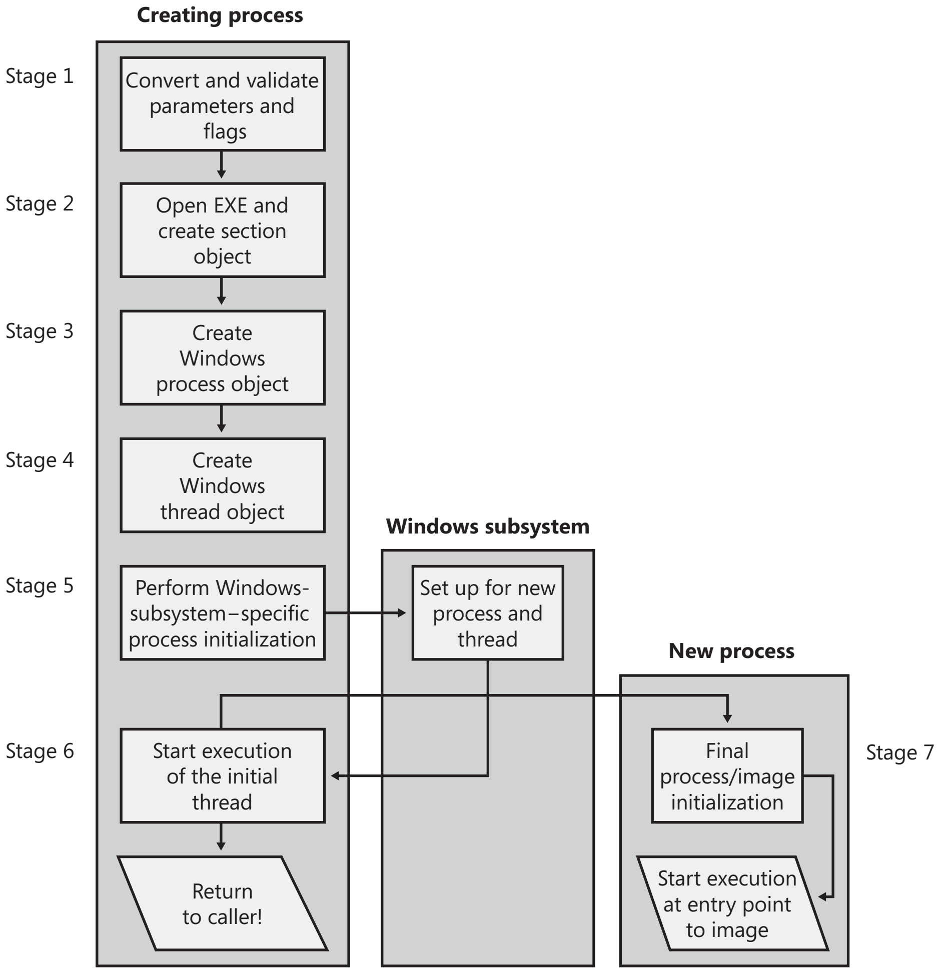

Creating process
Stage 1
Convert and validate parameters and flags
Stage 2
Open EXE and create section object
Stage 3
Create Windows process object
Stage 4
Create Windows thread object
Windows subsystem
Stage 5
Perform Windows-subsystem–specific process initialization
Set up for new process and thread
New process
Stage 6
Start execution of the initial thread
Final process/image initialization
Stage 7
Return to caller!
Start execution at entry point to image

FIGURE 5-5 The main stages of process creation

Stage 1: Converting and Validating Parameters and Flags

Before opening the executable image to run, *CreateProcess* performs the following steps:

- In *CreateProcess*, the priority class for the new process is specified as independent bits in the *CreationFlags* parameter. Thus, you can specify more than one priority class for a single *CreateProcess* call. Windows resolves the question of which priority class to assign to the process by choosing the lowest-priority class set.
- If no priority class is specified for the new process, the priority class defaults to Normal unless the priority class of the process that created it is Idle or Below Normal, in which case the priority class of the new process will have the same priority as the creating class.

- If a Real-time priority class is specified for the new process and the process's caller doesn't have the Increase Scheduling Priority privilege, the High priority class is used instead. In other words, *CreateProcess* doesn't fail just because the caller has insufficient privileges to create the process in the Real-time priority class; the new process just won't have as high a priority as Real-time.
- All windows are associated with desktops, the graphical representation of a workspace. If no desktop is specified in *CreateProcess*, the process is associated with the caller's current desktop.
- If the process is part of a job object, but the creation flags requested a separate virtual DOS machine (VDM), the flag is ignored.
- If the caller is sending a handle to a monitor as an output handle instead of a console handle, standard handle flags are ignored.
- If the creation flags specify that the process will be debugged, Kernel32 initiates a connection to the native debugging code in Ntdll.dll by calling *DbgUiConnectToDbg* and gets a handle to the debug object from the thread environment block (TEB) once the function returns.
- Kernel32.dll sets the default hard error mode if the creation flags specified one.
- The user-specified attribute list is converted from Windows subsystem format to native format, and internal attributes are added to it.

Note The attribute list passed on a *CreateProcess* call permits passing back to the caller information beyond a simple status code, such as the TEB address of the initial thread or information on the image section. This is necessary for protected processes since the parent cannot query this information after the child is created.

Once these steps are completed, *CreateProcess* will perform the initial call to *NtCreateUserProcess* to attempt creation of the process. Because Kernel32.dll has no idea at this point whether the application image name is a real Windows application, or if it might be a POSIX, 16-bit, or DOS application, the call may fail, at which point *CreateProcess* will look at the error reason and attempt to correct the situation.

Stage 2: Opening the Image to Be Executed

As illustrated in Figure 5-6, the first stage in *NtCreateUserProcess* is to find the appropriate Windows image that will run the executable file specified by the caller and to create a section object to later map it into the address space of the new process. If the call failed for any reason, it will return to *CreateProcess* with a failure state (see Table 5-6) that will cause *CreateProcess* to attempt execution again.

If the executable file specified is a Windows .exe, *NtCreateUserProcess* will try to open the file and create a section object for it. The object isn't mapped into memory yet, but it is opened.

Just because a section object has been successfully created doesn't mean that the file is a valid Windows image, however; it could be a DLL or a POSIX executable. If the file is a POSIX executable, the image to be run changes to Posix.exe, and *CreateProcess* restarts from the beginning of Stage 1. If the file is a DLL, *CreateProcess* fails.

Now that *NtCreateUserProcess* has found a valid Windows executable image, as part of the process creation code described in Stage 3 it looks in the registry under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options to see whether a subkey with the file name and extension of the executable image (but without the directory and path information—for example, Image.exe) exists there. If it does, *PspAllocateProcess* looks for a value named Debugger for that key. If this value is present, the image to be run becomes the string in that value and *CreateProcess* restarts at Stage 1.

Tip You can take advantage of this process creation behavior and debug the startup code of Windows services processes before they start rather than attach the debugger after starting a service, which doesn't allow you to debug the startup code.

On the other hand, if the image is not a Windows .exe (for example, if it's an MS-DOS, Win16, or a POSIX application), *CreateProcess* goes through a series of steps to find a Windows *support image* to run it. This process is necessary because non-Windows applications aren't run directly—Windows instead uses one of a few special support images that in turn are responsible for actually running the non-Windows program. For example, if you attempt to run a POSIX application, *CreateProcess* identifies it as such and changes the image to be run to the Windows executable file Posix.exe. If you attempt to run an MS-DOS or a Win16 executable, the image to be run becomes the Windows executable Ntvdm.exe. In short, you can't directly create a process that is *not* a Windows process. If Windows can't find a way to resolve the activated image as a Windows process (as shown in Table 5-6), *CreateProcess* fails.

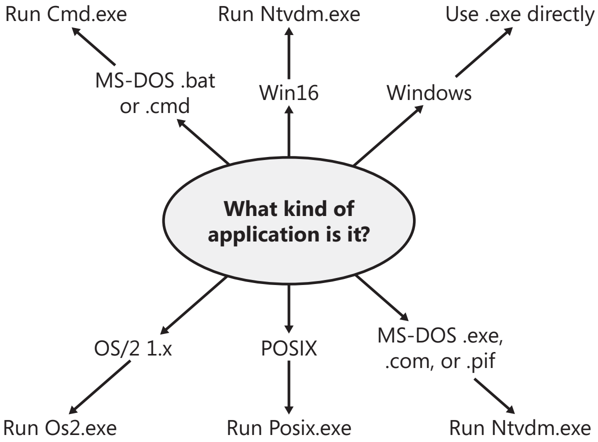


FIGURE 5-6 Choosing a Windows image to activate

TABLE 5-6 Decision Tree for Stage 1 of *CreateProcess*

If the Image . . .	Create State Code	This Image Will Run . . .	. . . and This Will Happen
Is a POSIX executable file	*PsCreateSuccess*	Posix.exe	*CreateProcess* restarts Stage 1.
Is an MS-DOS application with an .exe, a .com, or a .pif extension	*PsCreateFailOnSectionCreate*	Ntvdm.exe	*CreateProcess* restarts Stage 1.
Is a Win16 application	*PsCreateFailOnSectionCreate*	Ntvdm.exe	*CreateProcess* restarts Stage 1.
Is a Win64 application on a 32-bit system (or a PPC, MIPS, or Alpha Binary)	*PsCreateFailMachineMismatch*	N/A	*CreateProcess* will fail.
Has a Debugger key with another image name	*PsCreateFailExeName*	Name specified in the Debugger key	*CreateProcess* restarts Stage 1.
Is an invalid or damaged Windows EXE	*PsCreateFailExeFormat*	N/A	*CreateProcess* will fail.
Cannot be opened	*PsCreateFailOnFileOpen*	N/A	*CreateProcess* will fail.
Is a command procedure (application with a .bat or a .cmd extension)	*PsCreateFailOnSectionCreate*	Cmd.exe	*CreateProcess* restarts Stage 1.

Specifically, the decision tree that *CreateProcess* goes through to run an image is as follows:

- If the image is an MS-DOS application with an .exe, a .com, or a .pif extension, a message is sent to the Windows subsystem to check whether an MS-DOS support process (Ntvdm.exe, specified in the registry value HKLM\SYSTEM\CurrentControlSet\Control\WOW\cmdline) has already been created for this session. If a support process has been created, it is used to run the MS-DOS application. (The Windows subsystem sends the message to the VDM [Virtual DOS Machine] process to run the new image.) Then *CreateProcess* returns. If a support process hasn't been created, the image to be run changes to Ntvdm.exe and *CreateProcess* restarts at Stage 1.
- If the file to run has a .bat or a .cmd extension, the image to be run becomes Cmd.exe, the Windows command prompt, and *CreateProcess* restarts at Stage 1. (The name of the batch file is passed as the first parameter to Cmd.exe.)
- If the image is a Win16 (Windows 3.1) executable, *CreateProcess* must decide whether a new VDM process must be created to run it or whether it should use the default sessionwide shared VDM process (which might not yet have been created). The *CreateProcess* flags CREATE_SEPARATE_WOW_VDM and CREATE_SHARED_WOW_VDM control this decision. If these flags aren't specified, the registry value HKLM\SYSTEM\CurrentControlSet\Control\WOW\DefaultSeparateVDM dictates the default behavior.

If the application is to be run in a separate VDM, the image to be run changes to the value of HKLM\SYSTEM\CurrentControlSet\Control\WOW\wowcmdline and *CreateProcess* restarts at Stage 1. Otherwise, the Windows subsystem sends a message to see whether the shared VDM process exists and can be used. (If the VDM process is running on a different desktop or isn't running under the same security as the caller, it can't be used and a new VDM process must be created.) If a shared VDM process can be used, the Windows subsystem sends a message to it to run the new image and *CreateProcess* returns. If the VDM process hasn't yet been created (or if it exists but can't be used), the image to be run changes to the VDM support image and *CreateProcess* restarts at Stage 1.

Stage 3: Creating the Windows Executive Process Object (*PspAllocateProcess*)

At this point, *NtCreateUserProcess* has opened a valid Windows executable file and created a section object to map it into the new process address space. Next it creates a Windows executive process object to run the image by calling the internal system function *PspAllocateProcess*. Creating the executive process object (which is done by the creating thread) involves the following substages:

- Setting up the EPROCESS block
- Creating the initial process address space
- Initializing the kernel process block (KPROCESS)
- Setting up the PEB
- Concluding the setup of the process address space (which includes initializing the working set list and virtual address space descriptors and mapping the image into address space)

Note The only time there won't be a parent process is during system initialization. After that point, a parent process is always required to provide a security context for the new process.

Stage 3A: Setting Up the EPROCESS Block

This substage involves the following steps:

1. Allocate and initialize the Windows EPROCESS block.
2. Inherit the Windows device namespace (including the definition of drive letters, COM ports, and so on).

3. Inherit the process affinity mask and page priority from the parent process. If there is no parent process, the default page priority (5) is used, and an affinity mask of all processors (*KeActiveProcessors*) is used.
4. Set the new process's quota block to the address of its parent process's quota block, and increment the reference count for the parent's quota block. If the process was created through *CreateProcessAsUser*, this step won't occur.
5. The process minimum and maximum working set size are set to the values of *PspMinimumWorkingSet* and *PspMaximumWorkingSet*, respectively. These values can be overridden if performance options were specified in the PerfOptions key part of Image File Execution Options, in which case the maximum working set is taken from there.
6. Store the parent process's process ID in the *InheritedFromUniqueProcessId* field in the new process object.
7. Attach the process to the session of the parent process.
8. Initialize the KPROCESS part of the process object. (See Stage 3C.)
9. Create the process's primary access token (a duplicate of its parent's primary token). New processes inherit the security profile of their parents. If the *CreateProcessAsUser* function is being used to specify a different access token for the new process, the token is then changed appropriately.
10. The process handle table is initialized. If the inherit handles flag is set for the parent process, any inheritable handles are copied from the parent's object handle table into the new process. (For more information about object handle tables, see Chapter 3.) A process attribute can also be used to specify only a subset of handles, which is useful when you are using *CreateProcessAsUser* to restrict which objects should be inherited by the child process.
11. If performance options were specified through the PerfOptions key, these are now applied. The PerfOptions key includes overrides for the working set limit, I/O priority, page priority, and CPU priority class of the process.
12. The process priority class and quantum are computed and set.
13. Set the new process's exit status to STATUS_PENDING.

Stage 3B: Creating the Initial Process Address Space

The initial process address space consists of the following pages:

- Page directory (and it's possible there'll be more than one for systems with page tables more than two levels, such as x86 systems in PAE mode or 64-bit systems)
- Hyperspace page
- Working set list

To create these three pages, the following steps are taken:

1. Page table entries are created in the appropriate page tables to map the initial pages.
2. The number of pages is deducted from the kernel variable *MmTotalCommittedPages* and added to *MmProcessCommit*.
3. The systemwide default process minimum working set size (*PsMinimumWorkingSet*) is deducted from *MmResidentAvailablePages*.
4. The page table pages for the nonpaged portion of system space and the system cache are mapped into the process.

Stage 3C: Creating the Kernel Process Block

The next stage of *PspAllocateProcess* is the initialization of the KPROCESS block. This work is performed by *KeInitializeProcess*, which contains:

- A pointer to a list of kernel threads. (The kernel has no knowledge of handles, so it bypasses the object table.)
- A pointer to the process's page table directory (which is used to keep track of the process's virtual address space).
- The total time the process's threads have executed.
- The number of clock cycles the process's threads have consumed.
- The process's default base-scheduling priority (which starts as Normal, or 8, unless the parent process was set to Idle or Below Normal, in which case the setting is inherited).
- The default processor affinity for the threads in the process.
- The process swapping state (resident, out-swapped, or in transition).
- The NUMA ideal node (initially set to 0).
- The thread seed, based on the ideal processor that the kernel has chosen for this process (which is based on the previously created process's ideal processor, effectively randomizing this in a round-robin manner). Creating a new process will update the seed in *KeNodeBlock* (the initial NUMA node block) so that the next new process will get a different ideal processor seed.
- The initial value (or reset value) of the process default quantum (which is described in more detail in the "Thread Scheduling" section later in the chapter), which is hard-coded to 6 until it is initialized later (by *PspComputeQuantumAndPriority)*.

Note The default initial quantum differs between Windows client and server systems. For more information on thread quantums, turn to their discussion in the section "Thread Scheduling."

Stage 3D: Concluding the Setup of the Process Address Space

Setting up the address space for a new process is somewhat complicated, so let's look at what's involved one step at a time. To get the most out of this section, you should have some familiarity with the internals of the Windows memory manager, which are described in Chapter 9.

- The virtual memory manager sets the value of the process's last trim time to the current time. The working set manager (which runs in the context of the balance set manager system thread) uses this value to determine when to initiate working set trimming.
- The memory manager initializes the process's working set list—page faults can now be taken.
- The section (created when the image file was opened) is now mapped into the new process's address space, and the process section base address is set to the base address of the image.
- Ntdll.dll is mapped into the process.

Note POSIX processes clone the address space of their parents, so they don't have to go through these steps to create a new address space. In the case of POSIX applications, the new process's section base address is set to that of its parent process and the parent's PEB is cloned for the new process.

Stage 3E: Setting Up the PEB

NtCreateUserProcess calls *MmCreatePeb,* which first maps the systemwide national language support (NLS) tables into the process's address space. It next calls *MiCreatePebOrTeb* to allocate a page for the PEB and then initializes a number of fields, which are described in Table 5-7.

TABLE 5-7 Initial Values of the Fields of the PEB

Field	Initial Value
ImageBaseAddress	Base address of section
NumberOfProcessors	*KeNumberProcessors* kernel variable
NtGlobalFlag	*NtGlobalFlag* kernel variable
CriticalSectionTimeout	*MmCriticalSectionTimeout* kernel variable
HeapSegmentReserve	*MmHeapSegmentReserve* kernel variable
HeapSegmentCommit	*MmHeapSegmentCommit* kernel variable
HeapDeCommitTotalFreeThreshold	*MmHeapDeCommitTotalFreeThreshold* kernel variable
HeapDeCommitFreeBlockThreshold	*MmHeapDeCommitFreeBlockThreshold* kernel variable
NumberOfHeaps	0

Field	Initial Value
MaximumNumberOfHeaps	(Size of a page – size of a PEB) / 4
ProcessHeaps	First byte after PEB
MinimumStackCommit	*MmMinimumStackCommitInBytes* kernel variable
ImageProcessAffinityMask	*KeActiveProcessors or* 1 << *MmRotatingUniprocessorNumber* kernel variable (for uniprocessor-only images)
SessionId	Result of *MmGetSessionId*
ImageSubSystem	OptionalHeader.Subsystem
ImageSubSystemMajorVersion	OptionalHeader.MajorSubsystemVersion
ImageSubSystemMinorVersion	OptionalHeader.MinorSubsystemVersion
OSMajorVersion	*NtMajorVersion* kernel variable
OSMinorVersion	*NtMinorVersion* kernel variable
OSBuildNumber	*NtBuildNumber* kernel variable & 0x3FFF, combined with *CmNtCSDVersion* for service packs
OSPlatformId	2

However, if the image file specifies explicit Windows version or affinity values, this information replaces the initial values shown in Table 5-7. The mapping from image information fields to PEB fields is described in Table 5-8.

TABLE 5-8 Windows Replacements for Initial PEB Values

Field Name	Value Taken from Image Header
OSMajorVersion	OptionalHeader.Win32VersionValue & 0xFF
OSMinorVersion	(OptionalHeader.Win32VersionValue >> 8) & 0xFF
OSBuildNumber	(OptionalHeader.Win32VersionValue >> 16) & 0x3FFF, combined with ImageLoadConfigDirectory.CSDVersion
OSPlatformId	(OptionalHeader.Win32VersionValue >> 30) ^ 0x2
ImageProcessAffinityMask	ImageLoadConfigDirectory.ProcessAffinityMask

If the image header characteristics IMAGE_FILE_UP_SYSTEM_ONLY flag is set (indicating that the image can run only on a uniprocessor system), a single CPU is chosen for all the threads in this new process to run on. The selection process is performed by simply cycling through the available processors—each time this type of image is run, the next processor is used. In this way, these types of images are spread evenly across the processors.

If the image specifies an explicit processor affinity mask (for example, a field in the configuration header), this value is copied to the PEB and later set as the default process affinity mask.

Stage 3F: Completing the Setup of the Executive Process Object (*PspInsertProcess*)

Before the handle to the new process can be returned, a few final setup steps must be completed, which are performed by *PspInsertProcess* and its helper functions:

1. If systemwide auditing of processes is enabled (either as a result of local policy settings or group policy settings from a domain controller), the process's creation is written to the Security event log.
2. If the parent process was contained in a job, the job is recovered from the job level set of the parent and then bound to the session of the newly created process. Finally, the new process is added to the job.
3. *PspInsertProcess* inserts the new process block at the end of the Windows list of active processes (*PsActiveProcessHead*).
4. The process debug port of the parent process is copied to the new child process, unless the NoDebugInherit flag is set (which can be requested when creating the process). If a debug port was specified, it is attached to the new process at this time.
5. Finally, *PspInsertProcess* notifies any registered callback routines, creates a handle for the new process by calling *ObOpenObjectByPointer,* and then returns this handle to the caller.

Stage 4: Creating the Initial Thread and Its Stack and Context

At this point, the Windows executive process object is completely set up. It still has no thread, however, so it can't do anything yet. It's now time to start that work. Normally, the *PspCreateThread* routine is responsible for all aspects of thread creation and is called by *NtCreateThread* when a new thread is being created. However, because the initial thread is created internally by the kernel without user-mode input, the two helper routines that *PspCreateThread* relies on are used instead: *PspAllocateThread* and *PspInsertThread*.

PspAllocateThread handles the actual creation and initialization of the executive thread object itself, while *PspInsertThread* handles the creation of the thread handle and security attributes and the call to *KeStartThread* to turn the executive object into a schedulable thread on the system. However, the thread won't do anything yet—it is created in a suspended state and isn't resumed until the process is completely initialized (as described in Stage 5).

Note The thread parameter (which can't be specified in *CreateProcess* but can be specified in *CreateThread*) is the address of the PEB. This parameter will be used by the initialization code that runs in the context of this new thread (as described in Stage 6).

PspAllocateThread performs the following steps:

1. An executive thread block (ETHREAD) is created and initialized.
2. Before the thread can execute, it needs a stack and a context in which to run, so these are set up. The stack size for the initial thread is taken from the image—there's no way to specify another size.
3. The thread environment block (TEB) is allocated for the new thread.
4. The user-mode thread start address is stored in the ETHREAD. This is the system-supplied thread startup function in Ntdll.dll (*RtlUserThreadStart*). The user's specified Windows start address is stored in the ETHREAD block in a different location so that debugging tools such as Process Explorer can query the information.
5. *KeInitThread* is called to set up the KTHREAD block. The thread's initial and current base priorities are set to the process's base priority, and its affinity and quantum are set to that of the process. This function also sets the initial thread ideal processor. (See the section "Ideal and Last Processor" for a description of how this is chosen.) *KeInitThread* next allocates a kernel stack for the thread and initializes the machine-dependent hardware context for the thread, including the context, trap, and exception frames. The thread's context is set up so that the thread will start in kernel mode in *KiThreadStartup*. Finally, *KeInitThread* sets the thread's state to Initialized and returns to *PspAllocateThread*.

Once that work is finished, *NtCreateUserProcess* will call *PspInsertThread* to perform the following steps:

1. A thread ID is generated for the new thread.
2. The thread count in the process object is incremented, and the thread is added into the process thread list.
3. The thread is put into a suspended state.
4. The object is inserted and any registered thread callbacks are called.
5. The handle is created with *ObOpenObjectByName*.
6. The thread is readied for execution by calling *KeStartThread*.

Stage 5: Performing Windows Subsystem–Specific Post-Initialization

Once *NtCreateUserProcess* returns with a success code, all the necessary executive process and thread objects have been created. Kernel32.dll will now perform various operations related to Windows subsystem–specific operations to finish initializing the process.

First of all, various checks are made for whether Windows should allow the executable to run. These checks includes validating the image version in the header and checking whether Windows application certification has blocked the process (through a group policy). On specialized editions of Windows Server 2008, such as Windows Web Server 2008 and Windows HPC Server 2008, additional checks are made to see if the application imports any disallowed APIs.

If software restriction policies dictate, a restricted token is created for the new process. Afterward, the application compatibility database is queried to see if an entry exists in either the registry or system application database for the process. Compatibility shims will not be applied at this point—the information will be stored in the PEB once the initial thread starts executing (Stage 6).

At this point, Kernel32.dll sends a message to the Windows subsystem so that it can set up SxS information (see the end of this section for more information on side-by-side assemblies) such as manifest files, DLL redirection paths, and out-of-process execution for the new process. It also initializes the Windows subsystem structures for the process and initial thread. The message includes the following information:

- Process and thread handles
- Entries in the creation flags
- ID of the process's creator
- Flag indicating whether the process belongs to a Windows application (so that Csrss can determine whether or not to show the startup cursor)
- UI language Information
- DLL redirection and .local flags
- Manifest file information

The Windows subsystem performs the following steps when it receives this message:

1. *CsrCreateProcess* duplicates a handle for the process and thread. In this step, the usage count of the process and the thread is incremented from 1 (which was set at creation time) to 2.
2. If a process priority class isn't specified, *CsrCreateProcess* sets it according to the algorithm described earlier in this section.
3. The Csrss process block is allocated.
4. The new process's exception port is set to be the general function port for the Windows subsystem so that the Windows subsystem will receive a message when a second chance exception occurs in the process. (For further information on exception handling, see Chapter 3.)

5. The Csrss thread block is allocated and initialized.
6. *CsrCreateThread* inserts the thread in the list of threads for the process.
7. The count of processes in this session is incremented.
8. The process shutdown level is set to 0x280 (the default process shutdown level—see *SetProcessShutdownParameters* in the MSDN Library documentation for more information).
9. The new process block is inserted into the list of Windows subsystem-wide processes.
10. The per-process data structure used by the kernel-mode part of the Windows subsystem (W32PROCESS structure) is allocated and initialized.
11. The application start cursor is displayed. This cursor is the familiar rolling doughnut shape—the way that Windows says to the user, "I'm starting something, but you can use the cursor in the meantime." If the process doesn't make a GUI call after 2 seconds, the cursor reverts to the standard pointer. If the process does make a GUI call in the allotted time, *CsrCreateProcess* waits 5 seconds for the application to show a window. After that time, *CsrCreateProcess* will reset the cursor again.

After Csrss has performed these steps, *CreateProcess* checks whether the process was run elevated (which means it was executed through *ShellExecute* and elevated by the AppInfo service after the consent dialog box was shown to the user). This includes checking whether the process was a setup program. If it was, the process's token is opened, and the virtualization flag is turned on so that the application is virtualized. (See the information on UAC and virtualization in Chapter 6.) If the application contained elevation shims or had a requested elevation level in its manifest, the process is destroyed and an elevation request is sent to the AppInfo service. (See Chapter 6 for more information on elevation.)

Note that most of these checks are not performed for protected processes; because these processes must have been designed for Windows Vista or later, there's no reason why they should require elevation, virtualization, or application compatibility checks and processing. Additionally, allowing mechanisms such as the shim engine to use its usual hooking and memory patching techniques on a protected process would result in a security hole if someone could figure how to insert arbitrary shims that modify the behavior of the protected process.

Stage 6: Starting Execution of the Initial Thread

At this point, the process environment has been determined, resources for its threads to use have been allocated, the process has a thread, and the Windows subsystem knows about the new process. Unless the caller specified the CREATE_ SUSPENDED flag, the initial thread is now resumed so that it can start running and perform the remainder of the process initialization work that occurs in the context of the new process (Stage 7).

Stage 7: Performing Process Initialization in the Context of the New Process

The new thread begins life running the kernel-mode thread startup routine *KiThreadStartup*. *KiThreadStartup* lowers the thread's IRQL level from DPC/dispatch level to APC level and then calls the system initial thread routine, *PspUserThreadStartup*. The user-specified thread start address is passed as a parameter to this routine.

First, this function sets the Locale ID and the ideal processor in the TEB, based on the information present in kernel-mode data structures, and then it checks if thread creation actually failed. Next it calls *DbgkCreateThread,* which checks if image notifications were sent for the new process. If they weren't, and notifications are enabled, an image notification is sent first for the process and then for the image load of Ntdll.dll. Note that this is done in this stage rather than when the images were first mapped, because the process ID (which is required for the callouts) is not yet allocated at that time.

Once those checks are completed, another check is performed to see whether the process is a debuggee. If it is, then *PspUserThreadStartup* checks if the debugger notifications have already been sent for this process. If not, then a create process message is sent through the debug object (if one is present) so that the process startup debug event (CREATE_PROCESS_DEBUG_INFO) can be sent to the appropriate debugger process. This is followed by a similar thread startup debug event and by another debug event for the image load of Ntdll.dll. *DbgkCreateThread* then waits for the Windows subsystem to get the reply from the debugger (via the *ContinueDebugEvent* function).

Now that the debugger has been notified, *PspUserThreadStartup* looks at the result of the initial check on the thread's life. If it was killed on startup, the thread is terminated. This check is done after the debugger and image notifications to be sure that the kernel-mode and user-mode debuggers don't miss information on the thread, even if the thread never got a chance to run.

Otherwise, the routine checks whether application prefetching is enabled on the system and, if so, calls the prefetcher (and Superfetch) to process the prefetch instruction file (if it exists) and prefetch pages referenced during the first 10 seconds the last time the process ran. (For details on the prefetcher and Superfetch, see Chapter 9.)

PspUserThreadStartup then checks if the systemwide cookie in the SharedUserData structure has been set up yet. If it hasn't, it generates it based on a hash of system information such as the number of interrupts processed, DPC deliveries, and page faults. This systemwide cookie is used in the internal decoding and encoding of pointers, such as in the heap manager (for more information on heap manager security, see Chapter 9), to protect against certain classes of exploitation.

Finally, *PspUserThreadStartup* sets up the initial thunk context to run the image loader initialization routine (*LdrInitializeThunk* in Ntdll.dll), as well as the systemwide thread startup stub (*RtlUserThreadStart* in Ntdll.dll). These steps are done by editing the context of the thread in place and then issuing an *exit from system service* operation, which will load the specially crafted user context. The *LdrInitializeThunk* routine initializes the loader, heap manager, NLS tables, thread-local storage (TLS) and fiber-local storage (FLS) array, and critical section structures. It then loads any required DLLs and calls the DLL entry points with the DLL_PROCESS_ATTACH function code. (See the sidebar "Side-by-Side Assemblies" for a description of a mechanism Windows uses to address DLL versioning problems.)

Once the function returns, *NtContinue* will restore the new user context and return back to user mode—thread execution now truly starts.

RtlUserThreadStart will use the address of the actual image entry point and the start parameter and call the application. These two parameters have also already been pushed onto the stack by the kernel. This complicated series of events has two purposes. First of all, it allows the image loader inside Ntdll.dll to set up the process internally and behind the scenes so that other user-mode code can run properly (otherwise, it would have no heap, no thread local storage, and so on).

Second, having all threads begin in a common routine allows them to be wrapped in exception handling, so that when they crash, Ntdll.dll is aware of that and can call the unhandled exception filter inside Kernel32.dll. It is also able to coordinate thread exit on return from the thread's start routine and to perform various cleanup work. Application developers can also call *SetUnhandledExceptionFilter* to add their own unhandled exception handling code.

Side-by-Side Assemblies

In order to isolate DLLs distributed with applications from DLLs that ship with the operating system, Windows allows applications to use private copies of these core DLLs. To use a private copy of a DLL instead of the one in the system directory, an application's installation must include a file named *Application*.exe.local (where *Application* is the name of the application's executable), which directs the loader to first look for DLLs in that directory. Note that any DLLs that are loaded from the list of KnownDLLs (DLLs that are permanently mapped into memory) or that are loaded by those DLLs cannot be redirected using this mechanism.

To further address application and DLL compatibility while allowing sharing, Windows implements the concept of *shared assemblies*. An *assembly* consists of a group of resources, including DLLs, and an XML manifest file that describes the assembly and its contents. An application references an assembly through the existence of its own XML manifest. The manifest can be a file in the application's installation directory that has the same name as the application with ".manifest" appended (for example, application.exe.manifest), or it can be linked into the application as a resource. The manifest describes the application and its dependence on assemblies.

There are two types of assemblies: private and shared. The difference between the two is that shared assemblies are digitally signed so that corruption or modification of their contents can be detected. In addition, shared assemblies are stored under the \Windows\Winsxs directory, whereas private assemblies are stored in an application's installation directory. Thus, shared assemblies also have an associated catalog file (.cat) that contains its digital signature information. Shared assemblies can be "side-by-side" assemblies because multiple versions of a DLL can reside on a system simultaneously, with applications dependent on a particular version of a DLL always using that particular version.

An assembly's manifest file typically has a name that includes the name of the assembly, version information, some text that represents a unique signature, and the extension ".manifest". The manifests are stored in \Windows\Winsxs\Manifests, and the rest of the assembly's resources are stored in subdirectories of \Windows\Winsxs that have the same name as the corresponding manifest files, with the exception of the trailing .manifest extension.

An example of a shared assembly is version 6 of the Windows common controls DLL, comctl32.dll. Its manifest file is named \Windows\Winsxs\Manifests\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.0.0_x-ww_1382d70a.manifest. It has an associated catalog file (which is the same name with the .cat extension) and a subdirectory of Winsxs that includes comctl32.dll.

Version 6 of Comctl32.dll added integration with Windows themes, and because applications not written with theme support in mind might not appear correctly with the new DLL, it's available only to applications that explicitly reference the shared assembly containing it—the version of Comctl32.dll installed in \Windows\System32 is an instance of version 5.x, which is not theme aware. When an application loads, the loader looks for the application's manifest, and if one exists, loads the DLLs from the assemblies specified. DLLs not included in assemblies referenced in the manifest are loaded in the traditional way. Legacy applications, therefore, link against the version in \Windows\System32, whereas theme-aware applications can specify the new version in their manifest.

A final advantage that shared assemblies have is that a publisher can issue a publisher configuration, which can redirect all applications that use a particular assembly to use an updated version. Publishers would do this if they were preserving backward compatibility while addressing bugs. Ultimately, however, because of the flexibility inherent in the assembly model, an application could decide to override the new setting and continue to use an older version.

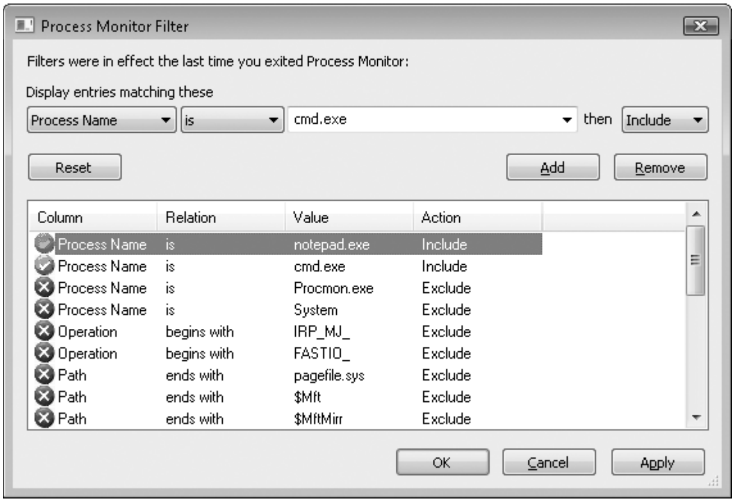
EXPERIMENT: Tracing Process Startup

Now that we've looked in detail at how a process starts up and the different operations required to begin executing an application, we're going to use Process Monitor to take a look at some of the file I/O and registry keys that are accessed during this process.

Although this experiment will not provide a complete picture of all the internal steps we've described, you'll be able to see several parts of the system in action, notably Prefetch and Superfetch, image file execution options and other compatibility checks, and the image loader's DLL mapping.

We're going to be looking at a very simple executable—Notepad.exe—and we will be launching it from a Command Prompt window (Cmd.exe). It's important that we look both at the operations inside Cmd.exe and those inside Notepad.exe. Recall that a lot of the user-mode work is performed by *CreateProcess*, which is called by the parent process before the kernel has created a new process object.

To set things up correctly, add two filters to Process Monitor: one for Cmd.exe, and one for Notepad.exe—these are the only two processes we want to include. It will be helpful to be sure that you don't have any currently running instances of these two processes so that you know you're looking at the right events. The filter window should look like this:



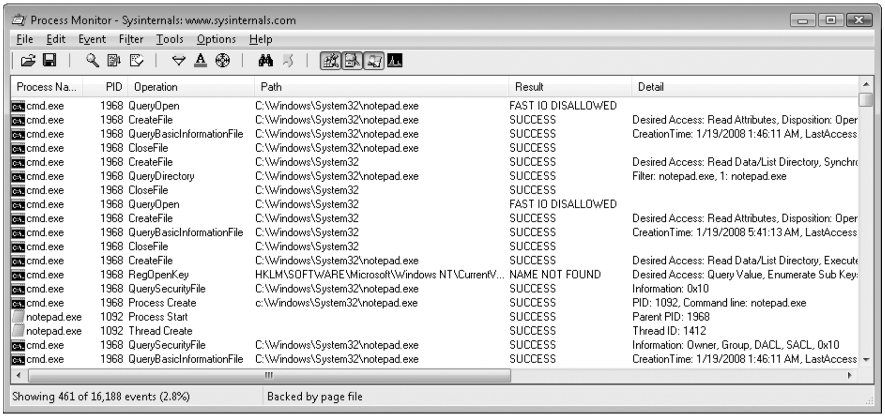
Filters were in effect the last time you exited Process Monitor:

Display entries matching these

Process Name | is | cmd.exe | then | Include

Column	Relation	Value	Action
✓ Process Name	is	notepad.exe	Include
✓ Process Name	is	cmd.exe	Include
☒ Process Name	is	Procmon.exe	Exclude
☒ Process Name	is	System	Exclude
☒ Operation	begins with	IRP_MJ_	Exclude
☒ Operation	begins with	FASTIO_	Exclude
☒ Path	ends with	pagefile.sys	Exclude
☒ Path	ends with	$Mft	Exclude
☒ Path	ends with	$MftMirr	Exclude

Next, make sure that event logging is currently disabled (clear File, Capture Events), and then start up the command prompt. Enable event logging (using the File menu again, or simply press CTRL+E or click the magnifying glass icon on the toolbar) and then enter **Notepad.exe** and press Enter. On a typical Windows Vista system, you should see anywhere between 500 and 1500 events appear. Go ahead and hide the Sequence and Time Of Day columns so that we can focus our attention on the columns of interest. Your window should look similar to the one shown next.

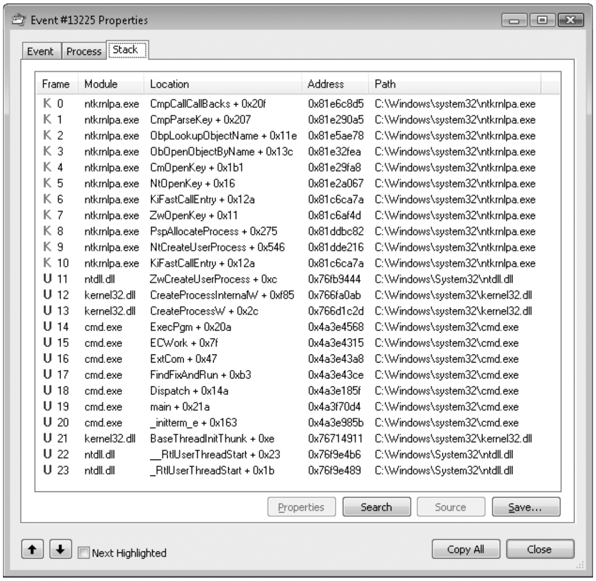

Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help

Process Na...	PID	Operation	Path	Result	Detail
cmd.exe	1968	QueryOpen	C:\Windows\System32\notepad.exe	FAST IO DISALLOWED	
cmd.exe	1968	CreateFile	C:\Windows\System32\notepad.exe	SUCCESS	Desired Access: Read Attributes, Disposition: Oper
cmd.exe	1968	QueryBasicInformationFile	C:\Windows\System32\notepad.exe	SUCCESS	CreationTime: 1/19/2008 1:46:11 AM, LastAccess
cmd.exe	1968	CloseFile	C:\Windows\System32\notepad.exe	SUCCESS	
cmd.exe	1968	CreateFile	C:\Windows\System32	SUCCESS	Desired Access: Read Data/List Directory, Synchro
cmd.exe	1968	QueryDirectory	C:\Windows\System32\notepad.exe	SUCCESS	Filter: notepad.exe, 1: notepad.exe
cmd.exe	1968	CloseFile	C:\Windows\System32	SUCCESS	
cmd.exe	1968	QueryOpen	C:\Windows\System32	FAST IO DISALLOWED	
cmd.exe	1968	CreateFile	C:\Windows\System32	SUCCESS	Desired Access: Read Attributes, Disposition: Oper
cmd.exe	1968	QueryBasicInformationFile	C:\Windows\System32	SUCCESS	CreationTime: 1/19/2008 5:41:13 AM, LastAccess
cmd.exe	1968	CloseFile	C:\Windows\System32	SUCCESS	
cmd.exe	1968	CreateFile	C:\Windows\System32\notepad.exe	SUCCESS	Desired Access: Read Data/List Directory, Execute
cmd.exe	1968	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentV...	NAME NOT FOUND	Desired Access: Query Value, Enumerate Sub Key:
cmd.exe	1968	QuerySecurityFile	C:\Windows\System32\notepad.exe	SUCCESS	Information: 0x10
cmd.exe	1968	Process Create	c:\Windows\System32\notepad.exe	SUCCESS	PID: 1092, Command line: notepad.exe
notepad.exe	1092	Process Start		SUCCESS	Parent PID: 1968
notepad.exe	1092	Thread Create		SUCCESS	Thread ID: 1412
cmd.exe	1968	QuerySecurityFile	C:\Windows\System32\notepad.exe	SUCCESS	Information: Owner, Group, DACL, SACL, 0x10
cmd.exe	1968	QueryBasicInformationFile	C:\Windows\System32\notepad.exe	SUCCESS	CreationTime: 1/19/2008 1:46:11 AM, LastAccess

Showing 461 of 16,188 events (2.8%) Backed by page file

Just as described in Stage 1 of the *CreateProcess* flow, one of the first things to notice is that just before the process is started and the first thread is created, Cmd.exe does a registry read at HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options. Because there were no image execution options associated with Notepad.exe, the process was created as is.

As with this and any other event in Process Monitor's log, you have the ability to see whether each part of the process creation flow was performed in user mode or kernel mode, and by which routines, by looking at the stack of the event. To do this, double-click on the RegOpenKey event mentioned and switch to the Stack tab. The following screen shows the standard stack on a 32-bit Windows Vista machine.



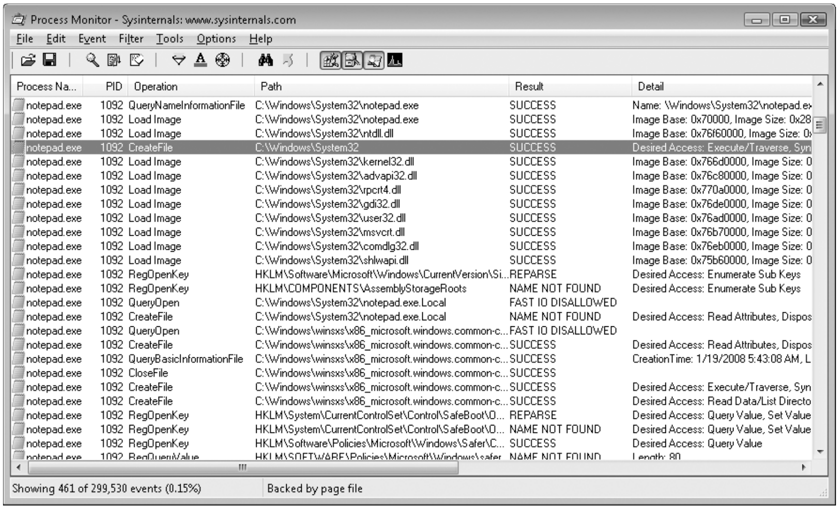
Frame	Module	Location	Address	Path
K 0	ntkrnlpa.exe	CmpCallCallBacks + 0x20f	0x81e6c8d5	C:\Windows\system32\ntkrnlpa.exe
K 1	ntkrnlpa.exe	CmpParseKey + 0x207	0x81e290a5	C:\Windows\system32\ntkrnlpa.exe
K 2	ntkrnlpa.exe	ObpLookupObjectName + 0x11e	0x81e5ae78	C:\Windows\system32\ntkrnlpa.exe
K 3	ntkrnlpa.exe	ObOpenObjectByName + 0x13c	0x81e32fea	C:\Windows\system32\ntkrnlpa.exe
K 4	ntkrnlpa.exe	CmOpenKey + 0x1b1	0x81e29fa8	C:\Windows\system32\ntkrnlpa.exe
K 5	ntkrnlpa.exe	NtOpenKey + 0x16	0x81e2a067	C:\Windows\system32\ntkrnlpa.exe
K 6	ntkrnlpa.exe	KiFastCallEntry + 0x12a	0x81c6ca7a	C:\Windows\system32\ntkrnlpa.exe
K 7	ntkrnlpa.exe	ZwOpenKey + 0x11	0x81c6af4d	C:\Windows\system32\ntkrnlpa.exe
K 8	ntkrnlpa.exe	PspAllocateProcess + 0x275	0x81ddbc82	C:\Windows\system32\ntkrnlpa.exe
K 9	ntkrnlpa.exe	NtCreateUserProcess + 0x546	0x81dde216	C:\Windows\system32\ntkrnlpa.exe
K 10	ntkrnlpa.exe	KiFastCallEntry + 0x12a	0x81c6ca7a	C:\Windows\system32\ntkrnlpa.exe
U 11	ntdll.dll	ZwCreateUserProcess + 0xc	0x76fb9444	C:\Windows\System32\ntdll.dll
U 12	kernel32.dll	CreateProcessInternalW + 0xf85	0x766fa0ab	C:\Windows\system32\kernel32.dll
U 13	kernel32.dll	CreateProcessW + 0x2c	0x766d1c2d	C:\Windows\system32\kernel32.dll
U 14	cmd.exe	ExecPgm + 0x20a	0x4a3e4568	C:\Windows\system32\cmd.exe
U 15	cmd.exe	ECWork + 0x7f	0x4a3e4315	C:\Windows\system32\cmd.exe
U 16	cmd.exe	ExtCom + 0x47	0x4a3e43a8	C:\Windows\system32\cmd.exe
U 17	cmd.exe	FindFixAndRun + 0xb3	0x4a3e43ce	C:\Windows\system32\cmd.exe
U 18	cmd.exe	Dispatch + 0x14a	0x4a3e185f	C:\Windows\system32\cmd.exe
U 19	cmd.exe	main + 0x21a	0x4a3f70d4	C:\Windows\system32\cmd.exe
U 20	cmd.exe	_initterm_e + 0x163	0x4a3e985b	C:\Windows\system32\cmd.exe
U 21	kernel32.dll	BaseThreadInitThunk + 0xe	0x76714911	C:\Windows\system32\kernel32.dll
U 22	ntdll.dll	__RtlUserThreadStart + 0x23	0x76f9e4b6	C:\Windows\System32\ntdll.dll
U 23	ntdll.dll	_RtlUserThreadStart + 0x1b	0x76f9e489	C:\Windows\System32\ntdll.dll

This stack shows that we have already reached the part of process creation performed in kernel mode (through *NtCreateUserProcess*) and that the helper routine *PspAllocateProcess* is responsible for this check.

Going down the list of events after the thread and process have been created, you will notice three groups of events. The first is a simple check for application compatibility flags, which will let the user-mode process creation code know if checks inside the application compatibility database are required through the shim engine.

This check is followed by multiple reads to Side-By-Side, Manifest, and MUI/Language keys, which are part of the assembly framework mentioned earlier. Finally, you may see file I/O to one or more .sdb files, which are the application compatibility databases on the system. This I/O is where additional checks are done to see if the shim engine needs to be invoked for this application. Since Notepad is a well behaved Microsoft program, it doesn't require any shims.

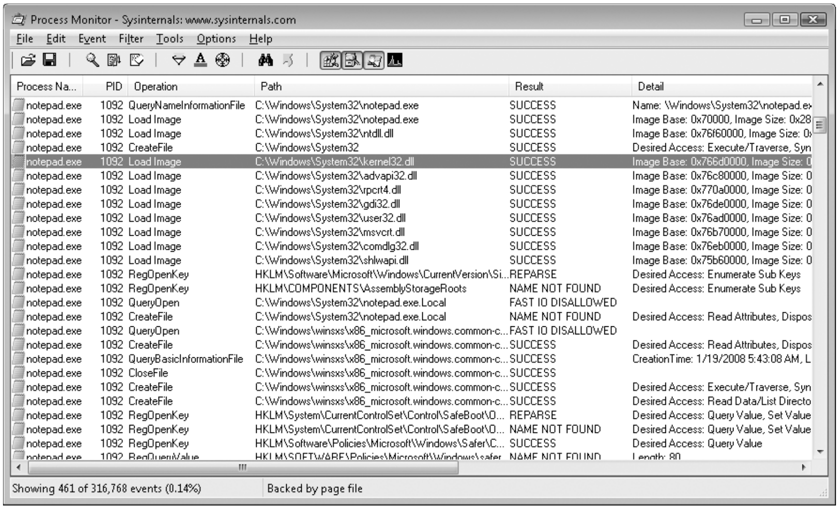
The following screen shows the next series of events, which happen inside the Notepad process itself. These are actions initiated by the user-mode thread startup wrapper in kernel mode, which performs the actions described earlier. The first two are the Notepad.exe and Ntdll.dll image load debug notification messages, which can only be generated now that code is running inside Notepad's process context and not the context for the command prompt.



Process Na...	PID	Operation	Path	Result	Detail
notepad.exe	1092	QueryNameInformationFile	C:\Windows\System32\notepad.exe	SUCCESS	Name: \Windows\System32\notepad.ex
notepad.exe	1092	Load Image	C:\Windows\System32\notepad.exe	SUCCESS	Image Base: 0x70000, Image Size: 0x28
notepad.exe	1092	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x76f60000, Image Size: 0x
notepad.exe	1092	CreateFile	C:\Windows\System32	SUCCESS	Desired Access: Execute/Traverse, Syn
notepad.exe	1092	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x766d0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\advapi32.dll	SUCCESS	Image Base: 0x76c80000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\rpcrt4.dll	SUCCESS	Image Base: 0x770a0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\gdi32.dll	SUCCESS	Image Base: 0x76de0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x76ad0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\msvcrt.dll	SUCCESS	Image Base: 0x76b70000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\comdlg32.dll	SUCCESS	Image Base: 0x76eb0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\shlwapi.dll	SUCCESS	Image Base: 0x75b60000, Image Size: 0
notepad.exe	1092	RegOpenKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Si...	REPARSE	Desired Access: Enumerate Sub Keys
notepad.exe	1092	RegOpenKey	HKLM\COMPONENTS\AssemblyStorageRoots	NAME NOT FOUND	Desired Access: Enumerate Sub Keys
notepad.exe	1092	QueryOpen	C:\Windows\System32\notepad.exe.Local	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\System32\notepad.exe.Local	NAME NOT FOUND	Desired Access: Read Attributes, Dispos
notepad.exe	1092	QueryOpen	C:\Windows\winsxs\x86_microsoft.windows.common-c...	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Attributes, Dispos
notepad.exe	1092	QueryBasicInformationFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	CreationTime: 1/19/2008 5:43:08 AM, L
notepad.exe	1092	CloseFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Execute/Traverse, Syn
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Data/List Directo
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\O...	REPARSE	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\O...	NAME NOT FOUND	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Safer\C...	SUCCESS	Desired Access: Query Value
notepad.exe	1092	RegQueryValue	HKLM\SOFTWARE\Policies\Microsoft\Windows\safer	NAME NOT FOUND	Length: 80

Showing 461 of 299,530 events (0.15%) Backed by page file

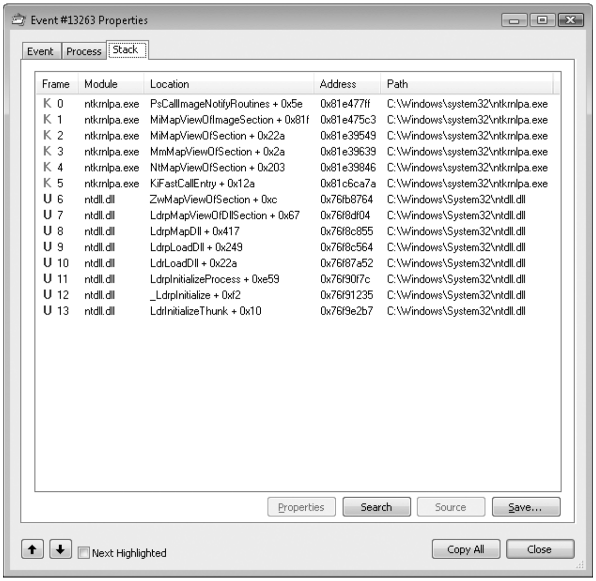
Next, the prefetcher kicks in, looking for a prefetch database file that has already been generated for Notepad. (For more information on the prefetcher, see Chapter 9). On a system where Notepad has already been run at least once, this database will exist, and the prefetcher will begin executing the commands specified inside it. If this is the case, scrolling down you will see multiple DLLs being read and queried. Unlike typical DLL loading, which is done by the user-mode image loader by looking at the import tables or when an application manually loads a DLL, these events are being generated by the prefetcher, which is already aware of the libraries that Notepad will require. Typical image loading of the DLLs required happens next, and you will see events similar to the ones shown here.


Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help

Process Na...	PID	Operation	Path	Result	Detail
notepad.exe	1092	QueryNameInformationFile	C:\Windows\System32\notepad.exe	SUCCESS	Name: \Windows\System32\notepad.ex
notepad.exe	1092	Load Image	C:\Windows\System32\notepad.exe	SUCCESS	Image Base: 0x70000, Image Size: 0x28
notepad.exe	1092	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x76f60000, Image Size: 0x
notepad.exe	1092	CreateFile	C:\Windows\System32	SUCCESS	Desired Access: Execute/Traverse, Syn
notepad.exe	1092	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x766d0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\advapi32.dll	SUCCESS	Image Base: 0x76c80000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\rpcrt4.dll	SUCCESS	Image Base: 0x770a0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\gdi32.dll	SUCCESS	Image Base: 0x76de0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x76ad0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\msvcrt.dll	SUCCESS	Image Base: 0x76b70000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\comdlg32.dll	SUCCESS	Image Base: 0x76eb0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\shlwapi.dll	SUCCESS	Image Base: 0x75b60000, Image Size: 0
notepad.exe	1092	RegOpenKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Si...	REPARSE	Desired Access: Enumerate Sub Keys
notepad.exe	1092	RegOpenKey	HKLM\COMPONENTS\AssemblyStorageRoots	NAME NOT FOUND	Desired Access: Enumerate Sub Keys
notepad.exe	1092	QueryOpen	C:\Windows\System32\notepad.exe.Local	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\System32\notepad.exe.Local	NAME NOT FOUND	Desired Access: Read Attributes, Dispos
notepad.exe	1092	QueryOpen	C:\Windows\winsxs\x86_microsoft.windows.common-c...	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Attributes, Dispos
notepad.exe	1092	QueryBasicInformationFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	CreationTime: 1/19/2008 5:43:08 AM, L
notepad.exe	1092	CloseFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Execute/Traverse, Syn
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Data/List Directo
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\O...	REPARSE	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\O...	NAME NOT FOUND	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Safer\C...	SUCCESS	Desired Access: Query Value
notepad.exe	1092	RegQueryValue	HKLM\SOFTWARE\Policies\Microsoft\Windows\safer...	NAME NOT FOUND	Length: 80

Showing 461 of 316,768 events (0.14%)   Backed by page file

These events are now being generated from code running inside user mode, which was called once the kernel-mode wrapper function finished its work. Therefore, these are the first events coming from *LdrpInitializeProcess*, which we mentioned is the internal system wrapper function for any new process, before the start address wrapper is called. You can confirm this on your own by looking at the stack of these events; for example, the kernel32.dll image load event, which is shown in the next screen.



Frame	Module	Location	Address	Path
K 0	ntkrnlpa.exe	PsCallImageNotifyRoutines + 0x5e	0x81e477ff	C:\Windows\system32\ntkrnlpa.exe
K 1	ntkrnlpa.exe	MiMapViewOfImageSection + 0x81f	0x81e475c3	C:\Windows\system32\ntkrnlpa.exe
K 2	ntkrnlpa.exe	MiMapViewOfSection + 0x22a	0x81e39549	C:\Windows\system32\ntkrnlpa.exe
K 3	ntkrnlpa.exe	MmMapViewOfSection + 0x2a	0x81e39639	C:\Windows\system32\ntkrnlpa.exe
K 4	ntkrnlpa.exe	NtMapViewOfSection + 0x203	0x81e39846	C:\Windows\system32\ntkrnlpa.exe
K 5	ntkrnlpa.exe	KiFastCallEntry + 0x12a	0x81c6ca7a	C:\Windows\system32\ntkrnlpa.exe
U 6	ntdll.dll	ZwMapViewOfSection + 0xc	0x76fb8764	C:\Windows\System32\ntdll.dll
U 7	ntdll.dll	LdrpMapViewOfDllSection + 0x67	0x76f8df04	C:\Windows\System32\ntdll.dll
U 8	ntdll.dll	LdrpMapDll + 0x417	0x76f8c855	C:\Windows\System32\ntdll.dll
U 9	ntdll.dll	LdrpLoadDll + 0x249	0x76f8c564	C:\Windows\System32\ntdll.dll
U 10	ntdll.dll	LdrLoadDll + 0x22a	0x76f87a52	C:\Windows\System32\ntdll.dll
U 11	ntdll.dll	LdrpInitializeProcess + 0xe59	0x76f90f7c	C:\Windows\System32\ntdll.dll
U 12	ntdll.dll	_LdrpInitialize + 0xf2	0x76f91235	C:\Windows\System32\ntdll.dll
U 13	ntdll.dll	LdrInitializeThunk + 0x10	0x76f9e2b7	C:\Windows\System32\ntdll.dll

Further events are generated by this routine and its associated helper functions until you finally reach events generated by the *WinMain* function inside Notepad, which is where code under the developer's control is now being executed. Describing in detail all the events and user-mode components that come into play during process execution would fill up this entire chapter, so exploration of any further events is left as an exercise for the reader.

Thread Internals

Now that we've dissected processes, let's turn our attention to the structure of a thread. Unless explicitly stated otherwise, you can assume that anything in this section applies to both user-mode threads and kernel-mode system threads (which are described in Chapter 2).

Data Structures

At the operating-system level, a Windows thread is represented by an executive thread (ETHREAD) block, which is illustrated in Figure 5-7. The ETHREAD block and the structures it points to exist in the system address space, with the exception of the thread environment block (TEB), which exists in the process address space (again, because user-mode components need to have access to it).

In addition, the Windows subsystem process (Csrss) also maintains a parallel structure for each thread created in a Windows subsystem application. Also, for threads that have called a Windows subsystem USER or GDI function, the kernel-mode portion of the Windows subsystem (Win32k.sys) maintains a per-thread data structure (called the W32THREAD structure) that the ETHREAD block points to.

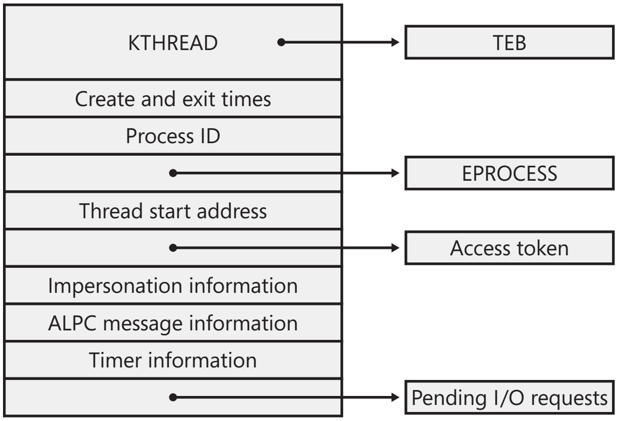


FIGURE 5-7 Structure of the executive thread block

Most of the fields illustrated in Figure 5-7 are self-explanatory. The first field is the kernel thread (KTHREAD) block. Following that are the thread identification information, the process identification information (including a pointer to the owning process so that its environment information can be accessed), security information in the form of a pointer to the access token and impersonation information, and finally, fields relating to ALPC messages and pending I/O requests. As you can see in Table 5-9, some of these key fields are covered in more detail elsewhere in this book. For more details on the internal structure of an ETHREAD block, you can use the kernel debugger *dt* command to display the format of the structure.

TABLE 5-9 **Key Contents of the Executive Thread Block**

Field Name	Value Taken from Image Header	Additional Information
KTHREAD	See Table 5-10.	
Thread time	Thread create and exit time information.	
Process identification	Process ID and pointer to EPROCESS block of the process that the thread belongs to.	
Start address	Address of thread start routine.	
Impersonation information	Access token and impersonation level (if the thread is impersonating a client).	Chapter 6
ALPC information	Message ID that the thread is waiting for and address of message.	Advanced local procedure calls (ALPC) (Chapter 3)
I/O information	List of pending I/O request packets (IRPs).	I/O system (Chapter 7)

Let’s take a closer look at two of the key thread data structures referred to in the preceding text: the KTHREAD block and the TEB. The KTHREAD block (also called the TCB, or thread control block) contains the information that the Windows kernel needs to access to perform thread scheduling and synchronization on behalf of running threads. Its layout is illustrated in Figure 5-8.

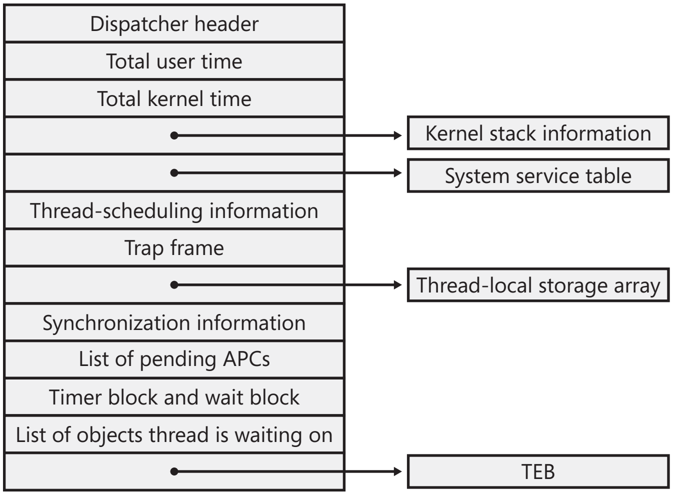

Dispatcher header
Total user time
Total kernel time
Kernel stack information
System service table
Thread-scheduling information
Trap frame
Thread-local storage array
Synchronization information
List of pending APCs
Timer block and wait block
List of objects thread is waiting on
TEB

FIGURE 5-8 Structure of the kernel thread block

The key fields of the KTHREAD block are described briefly in Table 5-10.

TABLE 5-10 Key Contents of the KTHREAD Block

Element	Description	Additional Reference
Dispatcher header	Because the thread is an object that can be waited on, it starts with a standard kernel dispatcher object header.	Kernel dispatcher objects (Chapter 3)
Execution time	Total user and kernel CPU time.	
Cycle time	Total CPU cycle time.	Thread scheduling
Pointer to kernel stack information	Base and upper address of the kernel stack.	Memory management (Chapter 9)
Pointer to system service table	Each thread starts out with this field service table pointing to the main system service table (*KeServiceDescriptorTable*). When a thread first calls a Windows GUI service, its system service table is changed to one that includes the GDI and USER services in Win32k.sys.	System service dispatching (Chapter 3)

Element	Description	Additional Reference
Scheduling information	Base and current priority, quantum target, quantum reset, affinity mask, ideal processor, deferred processor, next processor, scheduling state, freeze count, suspend count, adjust increment and adjust reason.	Thread scheduling
Wait blocks	The thread block contains four built-in wait blocks so that wait blocks don't have to be allocated and initialized each time the thread waits for something. (One wait block is dedicated to timers.)	Synchronization (Chapter 3)
Wait information	List of objects the thread is waiting for, wait reason, IRQL at the time of wait, result of the wait, and time at which the thread entered the wait state.	Synchronization (Chapter 3)
Mutant list	List of mutant objects the thread owns.	Synchronization (Chapter 3)
APC queues	List of pending user-mode and kernel-mode APCs, alerted flag, and flags to disable APCs.	Asynchronous procedure call (APC) interrupts (Chapter 3)
Timer block	Built-in timer block (also a corresponding wait block).	
Suspend APC and semaphore	Built-in APC and semaphore used when suspending and resuming a thread.	Synchronization (Chapter 3)
Queue	Pointer to queue object that the thread is associated with.	Synchronization (Chapter 3)
Gate	Pointer to gate object that the thread is waiting on.	Synchronization (Chapter 3)
Pointer to TEB	Thread ID, TLS and FLS information, PEB pointer, and Winsock, RPC, GDI, OpenGL, and other user-mode information.	

EXPERIMENT: Displaying ETHREAD and KTHREAD Structures

The ETHREAD and KTHREAD structures can be displayed with the *dt* command in the kernel debugger. The following output shows the format of an ETHREAD on a 32-bit system:

```
lkd> dt nt!_ethread
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1e0 CreateTime       : _LARGE_INTEGER
   +0x1e8 ExitTime         : _LARGE_INTEGER
   +0x1e8 KeyedWaitChain   : _LIST_ENTRY
   +0x1f0 ExitStatus       : Int4B
   +0x1f0 OfsChain         : Ptr32 Void
   +0x1f4 PostBlockList    : _LIST_ENTRY
   +0x1f4 ForwardLinkShadow : Ptr32 Void
   +0x1f8 StartAddress     : Ptr32 Void
   +0x1fc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1fc ReaperLink       : Ptr32 _ETHREAD
   +0x1fc KeyedWaitValue   : Ptr32 Void
   +0x1fc Win32StartParameter : Ptr32 Void
   +0x200 ActiveTimerListLock : Uint4B
   +0x204 ActiveTimerListHead : _LIST_ENTRY
   +0x20c Cid              : _CLIENT_ID
   +0x214 KeyedWaitSemaphore : _KSEMAPHORE
   +0x214 AlpcWaitSemaphore : _KSEMAPHORE
   +0x228 ClientSecurity   : _PS_CLIENT_SECURITY_CONTEXT
   +0x22c IrpList          : _LIST_ENTRY
   +0x234 TopLevelIrp      : Uint4B
   +0x238 DeviceToVerify   : Ptr32 _DEVICE_OBJECT
   +0x23c RateControlApc   : Ptr32 _PSP_RATE_APC
   +0x240 Win32StartAddress : Ptr32 Void
   +0x244 SparePtr0        : Ptr32 Void
   +0x248 ThreadListEntry  : _LIST_ENTRY
   +0x250 RundownProtect   : _EX_RUNDOWN_REF
   +0x254 ThreadLock       : _EX_PUSH_LOCK
   +0x258 ReadClusterSize  : Uint4B
   +0x25c MmLockOrdering   : Int4B
   +0x260 CrossThreadFlags : Uint4B
   +0x260 Terminated       : Pos 0, 1 Bit
   +0x260 ThreadInserted   : Pos 1, 1 Bit
   +0x260 HideFromDebugger : Pos 2, 1 Bit
   +0x260 ActiveImpersonationInfo : Pos 3, 1 Bit
   +0x260 SystemThread     : Pos 4, 1 Bit
   +0x260 HardErrorsAreDisabled : Pos 5, 1 Bit
   +0x260 BreakOnTermination : Pos 6, 1 Bit
   +0x260 SkipCreationMsg  : Pos 7, 1 Bit
   +0x260 SkipTerminationMsg : Pos 8, 1 Bit
   +0x260 CopyTokenOnOpen  : Pos 9, 1 Bit
   +0x260 ThreadIoPriority : Pos 10, 3 Bits
   +0x260 ThreadPagePriority : Pos 13, 3 Bits
   +0x260 RundownFail      : Pos 16, 1 Bit
   +0x264 SameThreadPassiveFlags : Uint4B
   +0x264 ActiveExWorker   : Pos 0, 1 Bit
   +0x264 ExWorkerCanWaitUser : Pos 1, 1 Bit
   +0x264 MemoryMaker      : Pos 2, 1 Bit
   +0x264 ClonedThread     : Pos 3, 1 Bit
   +0x264 KeyedEventInUse  : Pos 4, 1 Bit
   +0x264 RateApcState     : Pos 5, 2 Bits
   +0x264 SelfTerminate    : Pos 7, 1 Bit
   +0x268 SameThreadApcFlags : Uint4B
   +0x268 Spare            : Pos 0, 1 Bit
   +0x268 StartAddressInvalid : Pos 1, 1 Bit
   +0x268 EtwPageFaultCalloutActive : Pos 2, 1 Bit
   +0x268 OwnsProcessWorkingSetExclusive : Pos 3, 1 Bit
   +0x268 OwnsProcessWorkingSetShared : Pos 4, 1 Bit
   +0x268 OwnsSystemWorkingSetExclusive : Pos 5, 1 Bit
   +0x268 OwnsSystemWorkingSetShared : Pos 6, 1 Bit
   +0x268 OwnsSessionWorkingSetExclusive : Pos 7, 1 Bit
   +0x269 OwnsSessionWorkingSetShared : Pos 0, 1 Bit
   +0x269 OwnsProcessAddressSpaceExclusive : Pos 1, 1 Bit
   +0x269 OwnsProcessAddressSpaceShared : Pos 2, 1 Bit
   +0x269 SuppressSymbolLoad : Pos 3, 1 Bit
   +0x269 Prefetching      : Pos 4, 1 Bit
   +0x269 OwnsDynamicMemoryShared : Pos 5, 1 Bit
   +0x269 OwnsChangeControlAreaExclusive : Pos 6, 1 Bit
   +0x269 OwnsChangeControlAreaShared : Pos 7, 1 Bit
   +0x26a PriorityRegionActive : Pos 0, 4 Bits
   +0x26c CacheManagerActive : UChar
   +0x26d DisablePageFaultClustering : UChar
   +0x26e ActiveFaultCount : UChar
   +0x270 AlpcMessageId    : Uint4B
   +0x274 AlpcMessage      : Ptr32 Void
   +0x274 AlpcReceiveAttributeSet : Uint4B
   +0x278 AlpcWaitListEntry : _LIST_ENTRY
   +0x280 CacheManagerCount : Uint4B
```

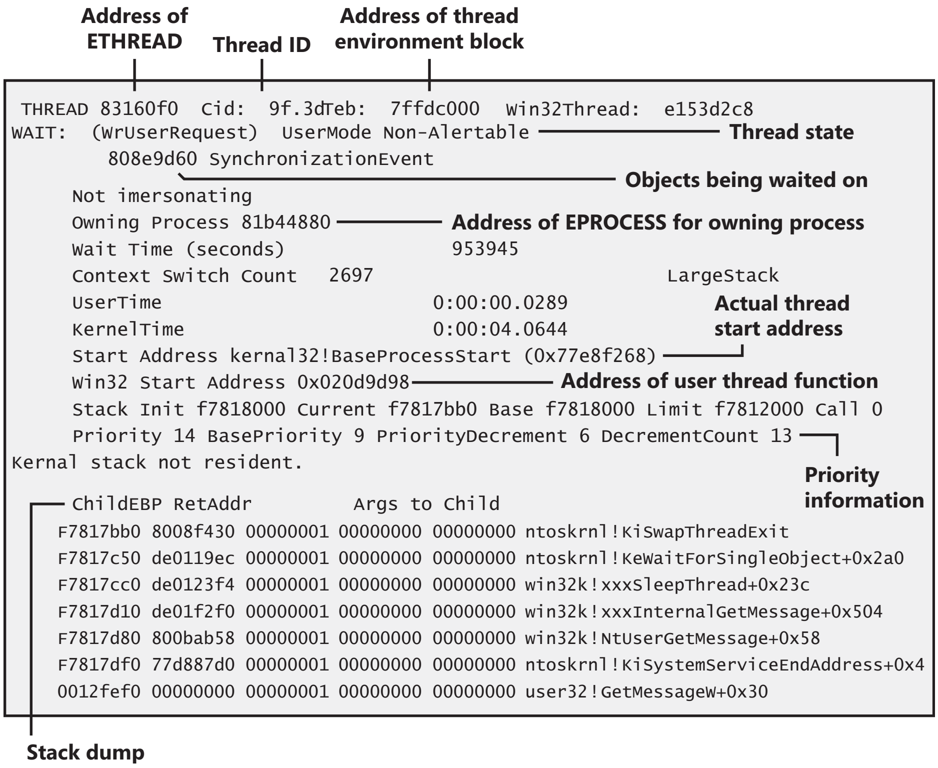
The KTHREAD can be displayed with a similar command:

```
lkd> dt nt!_kthread
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 CycleTime        : Uint8B
   +0x018 HighCycleTime    : Uint4B
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr32 Void
   +0x02c StackLimit       : Ptr32 Void
   +0x030 KernelStack      : Ptr32 Void
   +0x034 ThreadLock       : Uint4B
   +0x038 ApcState         : _KAPC_STATE
   +0x038 ApcStateFill     : [23] UChar
   +0x04f Priority         : Char
   +0x050 NextProcessor    : Uint2B
   +0x052 DeferredProcessor : Uint2B
   +0x054 ApcQueueLock     : Uint4B
   +0x058 ContextSwitches  : Uint4B
   +0x05c State            : UChar
   +0x05d NpxState         : UChar
   +0x05e WaitIrql         : UChar
   +0x05f WaitMode         : Char
   +0x060 WaitStatus       : Int4B
```

EXPERIMENT: Using the Kernel Debugger *!thread* Command

The kernel debugger *!thread* command dumps a subset of the information in the thread data structures. Some key elements of the information the kernel debugger displays can't be displayed by any utility: internal structure addresses; priority details; stack information; the pending I/O request list; and, for threads in a wait state, the list of objects the thread is waiting for.

To display thread information, use either the *!process* command (which displays all the thread blocks after displaying the process block) or the *!thread* command to dump a specific thread. The output of the thread information, along with some annotations of key fields, is shown here:



```
 THREAD 83160f0  Cid:  9f.3dTeb:  7ffdc000  Win32Thread:  e153d2c8
WAIT:  (WrUserRequest)  UserMode Non-Alertable
        808e9d60 SynchronizationEvent
     Not imersonating
     Owning Process 81b44880
     Wait Time (seconds)                 953945
     Context Switch Count   2697                              LargeStack
     UserTime                          0:00:00.0289
     KernelTime                        0:00:04.0644
     Start Address kernal32!BaseProcessStart (0x77e8f268)
     Win32 Start Address 0x020d9d98
     Stack Init f7818000 Current f7817bb0 Base f7818000 Limit f7812000 Call 0
     Priority 14 BasePriority 9 PriorityDecrement 6 DecrementCount 13
Kernal stack not resident.

     ChildEBP RetAddr         Args to Child
    F7817bb0 8008f430 00000001 00000000 00000000 ntoskrnl!KiSwapThreadExit
    F7817c50 de0119ec 00000001 00000000 00000000 ntoskrnl!KeWaitForSingleObject+0x2a0
    F7817cc0 de0123f4 00000001 00000000 00000000 win32k!xxxSleepThread+0x23c
    F7817d10 de01f2f0 00000001 00000000 00000000 win32k!xxxInternalGetMessage+0x504
    F7817d80 800bab58 00000001 00000000 00000000 win32k!NtUserGetMessage+0x58
    F7817df0 77d887d0 00000001 00000000 00000000 ntoskrnl!KiSystemServiceEndAddress+0x4
    0012fef0 00000000 00000001 00000000 00000000 user32!GetMessageW+0x30
```

Address of ETHREAD
Thread ID
Address of thread environment block
Thread state
Objects being waited on
Address of EPROCESS for owning process
Actual thread start address
Address of user thread function
Priority information
Stack dump

EXPERIMENT: Viewing Thread Information

The following output is the detailed display of a process produced by using the Tlist utility in the Debugging Tools for Windows. Notice that the thread list shows the "Win32StartAddr." This is the address passed to the *CreateThread* function by the application. All the other utilities, except Process Explorer, that show the thread start address show the actual start address (a function in Ntdll.dll), not the application-specified start address.

```
C:\> tlist winword
2400 WINWORD.EXE       WinInt5E_Chapter06.doc [Compatibility Mode] - Microsoft Word
   CWD:     C:\Users\Alex Ionescu\Documents\
   CmdLine: "C:\Program Files\Microsoft Office\Office12\WINWORD.EXE" /n /dde
   VirtualSize:   310656 KB   PeakVirtualSize:   343552 KB
   WorkingSetSize: 91548 KB   PeakWorkingSetSize:100788 KB
   NumberOfThreads: 6
   2456 Win32StartAddr:0x2f7f10cc LastErr:0x00000000 State:Waiting
   1452 Win32StartAddr:0x6882f519 LastErr:0x00000000 State:Waiting
   2464 Win32StartAddr:0x6b603850 LastErr:0x00000000 State:Waiting
   3036 Win32StartAddr:0x690dc17f LastErr:0x00000002 State:Waiting
   3932 Win32StartAddr:0x775cac65 LastErr:0x00000102 State:Waiting
   3140 Win32StartAddr:0x687d6ffd LastErr:0x000003f0 State:Waiting
 12.0.4518.1014 shp  0x2F7F0000  C:\Program Files\Microsoft Office\Office12\
     WINWORD.EXE
 6.0.6000.16386 shp  0x777D0000  C:\Windows\system32\Ntdll.dll
 6.0.6000.16386 shp  0x764C0000  C:\Windows\system32\kernel32.dll
         §          list of DLLs loaded in process
```

The TEB, illustrated in Figure 5-9, is the only data structure explained in this section that exists in the process address space (as opposed to the system space).

The TEB stores context information for the image loader and various Windows DLLs. Because these components run in user mode, they need a data structure writable from user mode. That's why this structure exists in the process address space instead of in the system space, where it would be writable only from kernel mode. You can find the address of the TEB with the kernel debugger *!thread* command.

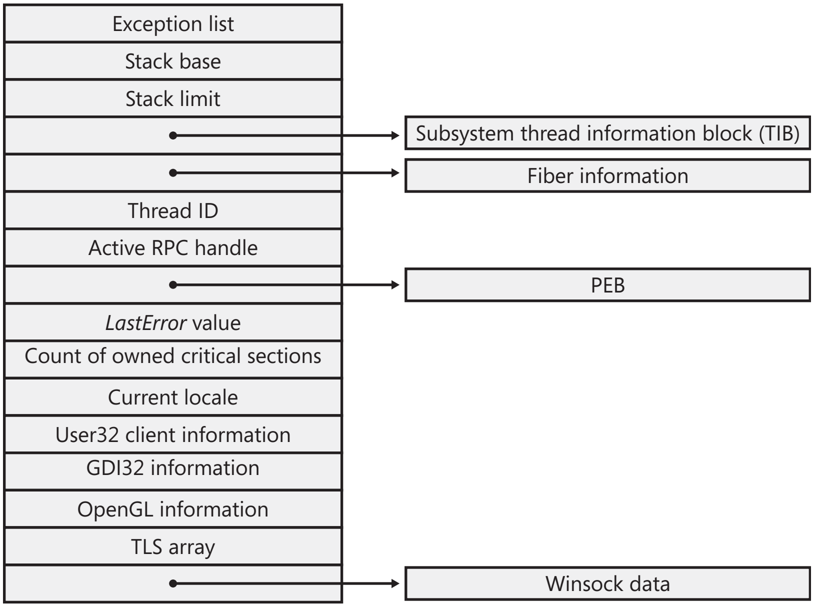


FIGURE 5-9 Fields of the thread environment block

EXPERIMENT: Examining the TEB

You can dump the TEB structure with the *!teb* command in the kernel debugger. The output looks like this:

```
kd> !teb
TEB at 7ffde000
    ExceptionList:        019e8e44
    StackBase:            019f0000
    StackLimit:           019db000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffde000
    EnvironmentPointer:   00000000
    ClientId:             00000bcc . 00000864
    RpcHandle:            00000000
    Tls Storage:          7ffde02c
    PEB Address:          7ffd9000
    LastErrorValue:       0
    LastStatusValue:      c0000139
    Count Owned Locks:    0
    HardErrorMode:        0
```

Kernel Variables

As with processes, a number of Windows kernel variables control how threads run. Table 5-11 shows the kernel-mode kernel variables that relate to threads.

TABLE 5-11 Thread-Related Kernel Variables

Variable	Type	Description
PspCreateThreadNotifyRoutine	Array of executive callback objects	Array of callback objects describing the routines to be called on thread creation and deletion (maximum of 64)
PspCreateThreadNotifyRoutineCount	32-bit integer	Count of registered thread-notification routines

Performance Counters

Most of the key information in the thread data structures is exported as performance counters, which are listed in Table 5-12. You can extract much information about the internals of a thread just by using the Reliability and Performance Monitor in Windows.

TABLE 5-12 Thread-Related Performance Counters

Object: Counter	Function
Process: Priority Base	Returns the current base priority of the process. This is the starting priority for threads created within this process.
Thread: % Privileged Time	Describes the percentage of time that the thread has run in kernel mode during a specified interval.
Thread: % Processor Time	Describes the percentage of CPU time that the thread has used during a specified interval. This count is the sum of % Privileged Time and % User Time.
Thread: % User Time	Describes the percentage of time that the thread has run in user mode during a specified interval.
Thread: Context Switches/Sec	Returns the number of context switches per second that the system is executing.
Thread: Elapsed Time	Returns the amount of CPU time (in seconds) that the thread has consumed.
Thread: ID Process	Returns the process ID of the thread's process.
Thread: ID Thread	Returns the thread's thread ID. This ID is valid only during the thread's lifetime because thread IDs are reused.
Thread: Priority Base	Returns the thread's current base priority. This number might be different from the thread's starting base priority.
Thread: Priority Current	Returns the thread's current dynamic priority.
Thread: Start Address	Returns the thread's starting virtual address (*Note*: This address will be the same for most threads.)

Object: Counter	Function
Thread: Thread State	Returns a value from 0 through 7 relating to the current state of the thread.
Thread: Thread Wait Reason	Returns a value from 0 through 19 relating to the reason why the thread is in a wait state.

Relevant Functions

Table 5-13 shows the Windows functions for creating and manipulating threads. This table doesn't include functions that have to do with thread scheduling and priorities—those are included in the section "Thread Scheduling" later in this chapter.

TABLE 5-13 Windows Thread Functions

Function	Description
CreateThread	Creates a new thread
CreateRemoteThread	Creates a thread in another process
OpenThread	Opens an existing thread
ExitThread	Ends execution of a thread normally
TerminateThread	Terminates a thread
IsThreadAFiber	Returns whether the current thread is a fiber
GetExitCodeThread	Gets another thread's exit code
GetThreadTimes	Returns timing information for a thread
QueryThreadCycleTime	Returns CPU clock cycle information for a thread
GetCurrentThread	Returns a pseudo handle for the current thread
GetCurrentProcessId	Returns the thread ID of the current thread
GetThreadId	Returns the thread ID of the specified thread
Get/SetThreadContext	Returns or changes a thread's CPU registers
GetThreadSelectorEntry	Returns another thread's descriptor table entry (applies only to x86 systems)

Birth of a Thread

A thread's life cycle starts when a program creates a new thread. The request filters down to the Windows executive, where the process manager allocates space for a thread object and calls the kernel to initialize the kernel thread block. The steps in the following list are taken inside the Windows *CreateThread* function in Kernel32.dll to create a Windows thread.

1. *CreateThread* converts the Windows API parameters to native flags and builds a native structure describing object parameters (OBJECT_ATTRIBUTES). See Chapter 3 for more information.

2. *CreateThread* builds an attribute list with two entries: client ID and TEB address. This allows *CreateThread* to receive those values once the thread has been created. (For more information on attribute lists, see the section "Flow of *CreateProcess"* earlier in this chapter.)
3. *NtCreateThreadEx* is called to create the user-mode context and probe and capture the attribute list. It then calls *PspCreateThread* to create a suspended executive thread object. For a description of the steps performed by this function, see the descriptions of Stage 3 and Stage 5 in the section "Flow of *CreateProcess*."
4. *CreateThread* allocates an activation stack for the thread used by side-by-side assembly support. It then queries the activation stack to see if it requires activation, and does so if needed. The activation stack pointer is saved in the new thread's TEB.
5. *CreateThread* notifies the Windows subsystem about the new thread, and the subsystem does some setup work for the new thread.
6. The thread handle and the thread ID (generated during step 3) are returned to the caller.
7. Unless the caller created the thread with the CREATE_SUSPENDED flag set, the thread is now resumed so that it can be scheduled for execution. When the thread starts running, it executes the steps described in the earlier section "Stage 7: Performing Process Initialization in the Context of the New Process" before calling the actual user's specified start address.

Examining Thread Activity

Examining thread activity is especially important if you are trying to determine why a process that is hosting multiple services is running (such as Svchost.exe, Dllhost.exe, or Lsass.exe) or why a process is hung.

There are several tools that expose various elements of the state of Windows threads: WinDbg (in user-process attach and kernel debugging mode), the Reliability and Performance Monitor, and Process Explorer. (The tools that show thread-scheduling information are listed in the section "Thread Scheduling.")

To view the threads in a process with Process Explorer, select a process and open the process properties (double-click on the process or click on the Process, Properties menu item). Then click on the Threads tab. This tab shows a list of the threads in the process and three columns of information. For each thread it shows the percentage of CPU consumed (based on the refresh interval configured), the number of context switches to the thread, and the thread start address. You can sort by any of these three columns.

New threads that are created are highlighted in green, and threads that exit are highlighted in red. (The highlight duration can be configured with the Options, Configure Highlighting menu item.) This might be helpful to discover unnecessary thread creation occurring in a process. (In general, threads should be created at process startup, not every time a request is processed inside a process.)

As you select each thread in the list, Process Explorer displays the thread ID, start time, state, CPU time counters, number of context switches, and the base and current priority. There is a Kill button, which will terminate an individual thread, but this should be used with extreme care.

The best way to measure actual CPU activity with Process Explorer is to add the clock cycle delta column, which uses the clock cycle counter designed for thread run-time accounting (as described later in this chapter). Because many threads run for such a short amount of time that they are seldom (if ever) the currently running thread when the clock interval timer interrupt occurs, they are not charged for much of their CPU time. The total number of clock cycles represents the actual number of processor cycles that each thread in the process accrued. It is independent of the clock interval timer's resolution because the count is maintained internally by the processor at each cycle and updated by Windows at each interrupt entry (a final accumulation is done before a context switch).

The thread start address is displayed in the form "*module*!*function*", where *module* is the name of the .exe or .dll. The function name relies on access to symbol files for the module. (See "Experiment: Viewing Process Details with Process Explorer" in Chapter 1.) If you are unsure what the module is, click the Module button. This opens an Explorer file properties window for the module containing the thread's start address (for example, the .exe or .dll).

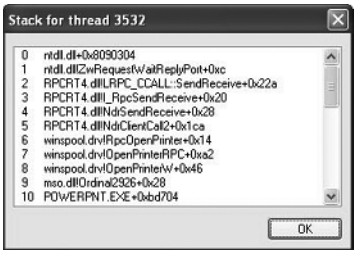
Note For threads created by the Windows *CreateThread* function, Process Explorer displays the function passed to *CreateThread*, not the actual thread start function. That is because all Windows threads start at a common thread startup wrapper function (*RtlUserThreadStart* in Ntdll.dll). If Process Explorer showed the actual start address, most threads in processes would appear to have started at the same address, which would not be helpful in trying to understand what code the thread was executing. However, if Process Explorer can't query the user-defined startup address (such as in the case of a protected process), it will show the wrapper function, so you will see all threads starting at *RtlUserThreadStart*.

However, the thread start address displayed might not be enough information to pinpoint what the thread is doing and which component within the process is responsible for the CPU consumed by the thread. This is especially true if the thread start address is a generic startup function (for example, if the function name does not indicate what the thread is actually doing). In this case, examining the thread stack might answer the question. To view the stack for a thread, double-click on the thread of interest (or select it and click the Stack button).

Process Explorer displays the thread's stack (both user and kernel, if the thread was in kernel mode).

> **Note** While the user-mode debuggers (WinDbg, Ntsd, and Cdb) permit you to attach to a process and display the user stack for a thread, Process Explorer shows both the user and kernel stack in one easy click of a button. You can also examine user and kernel thread stacks using WinDbg in local kernel debugging mode.

Viewing the thread stack can also help you determine why a process is hung. As an example, on one system, Microsoft Office PowerPoint was hanging for one minute on startup. To determine why it was hung, after starting PowerPoint, Process Explorer was used to examine the thread stack of the one thread in the process. The result is shown in Figure 5-10.

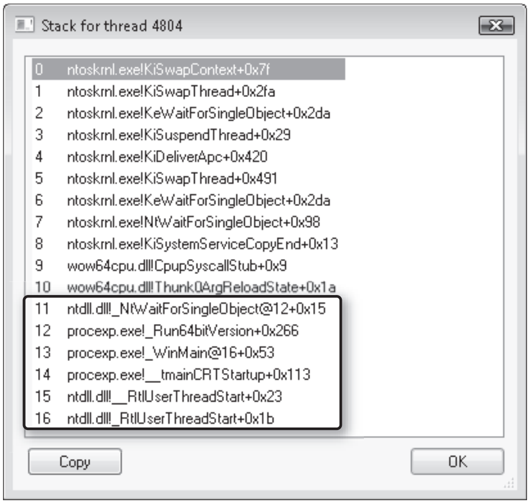


```
0   ntdll.dll+0x8090304
1   ntdll.dll!ZwRequestWaitReplyPort+0xc
2   RPCRT4.dll!LRPC_CCALL::SendReceive+0x22a
3   RPCRT4.dll!I_RpcSendReceive+0x20
4   RPCRT4.dll!NdrSendReceive+0x28
5   RPCRT4.dll!NdrClientCall2+0x1ca
6   winspool.drv!RpcOpenPrinter+0x14
7   winspool.drv!OpenPrinterRPC+0xa2
8   winspool.drv!OpenPrinterW+0x46
9   mso.dll!Ordinal2926+0x28
10  POWERPNT.EXE+0xbd704
```

FIGURE 5-10 Hung thread stack in PowerPoint

This thread stack shows that PowerPoint (line 10) called a function in Mso.dll (the central Microsoft Office DLL), which called the *OpenPrinterW* function in Winspool.drv (a DLL used to connect to printers). Winspool.drv then dispatched to a function *OpenPrinterRPC*, which then called a function in the RPC runtime DLL, indicating it was sending the request to a remote printer. So, without having to understand the internals of PowerPoint, the module and function names displayed on the thread stack indicate that the thread was waiting to connect to a network printer. On this particular system, there was a network printer that was not responding, which explained the delay starting PowerPoint. (Microsoft Office applications connect to all configured printers at process startup.) The connection to that printer was deleted from the user's system, and the problem went away.

Finally, when looking at 32-bit applications running on 64-bit systems as a Wow64 process (see Chapter 3 for more information on Wow64), Process Explorer shows both the 32-bit and 64-bit stack for threads. Because at the time of the system call proper, the thread has been switched to a 64-bit stack and context, simply looking at the thread's 64-bit stack would reveal only half the story—the 64-bit part of the thread, with Wow64's thunking code. So, when examining Wow64 processes, be sure to take into account both the 32-bit and 64-bit stacks. An example of a Wow64 thread inside Microsoft Office Word 2007 is shown in Figure 5-11. The stack frames highlighted in the box are the 32-bit stack frames from the 32-bit stack.



```
0    ntoskrnl.exe!KiSwapContext+0x7f
1    ntoskrnl.exe!KiSwapThread+0x2fa
2    ntoskrnl.exe!KeWaitForSingleObject+0x2da
3    ntoskrnl.exe!KiSuspendThread+0x29
4    ntoskrnl.exe!KiDeliverApc+0x420
5    ntoskrnl.exe!KiSwapThread+0x491
6    ntoskrnl.exe!KeWaitForSingleObject+0x2da
7    ntoskrnl.exe!NtWaitForSingleObject+0x98
8    ntoskrnl.exe!KiSystemServiceCopyEnd+0x13
9    wow64cpu.dll!CpupSyscallStub+0x9
10   wow64cpu.dll!Thunk0ArgReloadState+0x1a
11   ntdll.dll!_NtWaitForSingleObject@12+0x15
12   procexp.exe!_Run64bitVersion+0x266
13   procexp.exe!_WinMain@16+0x53
14   procexp.exe!__tmainCRTStartup+0x113
15   ntdll.dll!__RtlUserThreadStart+0x23
16   ntdll.dll!_RtlUserThreadStart+0x1b
```

FIGURE 5-11 Example Wow64 stack

Limitations on Protected Process Threads

As we discussed in the process internals section, protected processes have several limitations in terms of which access rights will be granted, even to the users with the highest privileges on the system. These limitations also apply to threads inside such a process. This ensures that the actual code running inside the protected process cannot be hijacked or otherwise affected through standard Windows functions, which require the access rights in Table 5-14.

TABLE 5-14 Thread Access Rights Denied for Threads Inside a Protected Process

Object: Access Mask	Function
Thread: THREAD_ALL_ACCESS	Disables full access to a thread inside a protected process.
Thread: THREAD_DIRECT_IMPERSONATION	Disables impersonating a thread inside a protected process.
Thread: THREAD_GET_CONTEXT, THREAD_SET_CONTEXT	Disables accessing and/or modifying the CPU context (registers and stack) of a thread inside a protected process.
Thread: THREAD_QUERY_INFORMATION	Disables querying all information on a thread inside a protected process. However, a new access right was added, THREAD_QUERY_LIMITED_INFORMATION, that grants limited access to information on the thread.

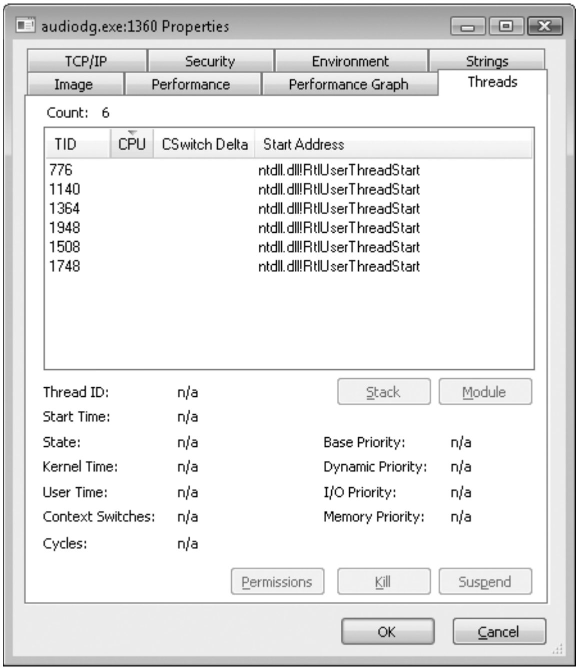
Object: Access Mask	Function
Thread: THREAD_SET_INFORMATION	Disables setting all information on a thread inside a protected process. However, a new access right was added, THREAD_SET_LIMITED_INFORMATION, that grants limited access to modifying information on the thread.
Thread: THREAD_SET_THREAD_TOKEN	Disables setting an impersonation token for a thread inside a protected process.
Thread: THREAD_TERMINATE	Disables terminating a thread inside a protected process. Note that terminating all threads atomically through process termination is allowed.

EXPERIMENT: Viewing Protected Process Thread Information

In the previous section, we took a look at how Process Explorer can be helpful in examining thread activity to determine the cause of potential system or application issues. This time, we'll use Process Explorer to look at a protected process and see how the different access rights being denied affect its ability and usefulness on such a process.

Find the Audiodg.exe service inside the process list. This is a process responsible for much of the core work behind the user-mode audio stack in Windows Vista, and it requires protection to ensure that high-definition decrypted audio content does not leak out to untrusted sources. Bring up the process properties view and take a look at the Image tab. Notice how the numbers for WS Private, WS Shareable, and WS Shared are 0, although the total Working Set is still displayed. This is an example of the THREAD_QUERY_INFORMATION versus THREAD_QUERY_LIMITED_INFORMATION rights.

More importantly, take a look at the Threads tab. As you can see here, Process Explorer is unable to show the Win32 thread start address and instead displays the standard thread start wrapper inside Ntdll.dll. If you try clicking on the Stack button, you'll get an error, because Process Explorer needs to read the virtual memory inside the protected process, which it can't do.



Count: 6

TID	CPU	CSwitch Delta	Start Address
776			ntdll.dll!RtlUserThreadStart
1140			ntdll.dll!RtlUserThreadStart
1364			ntdll.dll!RtlUserThreadStart
1948			ntdll.dll!RtlUserThreadStart
1508			ntdll.dll!RtlUserThreadStart
1748			ntdll.dll!RtlUserThreadStart

Thread ID: n/a
Start Time: n/a
State: n/a
Kernel Time: n/a
User Time: n/a
Context Switches: n/a
Cycles: n/a

Base Priority: n/a
Dynamic Priority: n/a
I/O Priority: n/a
Memory Priority: n/a

Finally, note that although the Base and Dynamic priorities are shown, the I/O and Memory priorities are not, another example of the limited versus full query information access right. As you try to kill a thread inside Audiodg.exe, notice yet another access denied error: recall the lack of THREAD_TERMINATE access shown earlier in Table 5-14.

Worker Factories (Thread Pools)

Worker factories refer to the internal mechanism used to implement user-mode thread pools. Prior to Windows Vista, the thread pool routines were completely implemented in user mode inside the Ntdll.dll library, and the Windows API provided various routines to call into the relevant routines, which provided waitable timers, wait callbacks, and automatic thread creation and deletion depending on the amount of work being done.

Note Information on the new thread pool API is available on MSDN at *http://msdn2.microsoft.com/en-us/library/ms686760.aspx*. It includes information on the APIs introduced and the APIs retired, as well as important differences in certain details of the way the two APIs are implemented.

In Windows Vista, the thread pool implementation in user mode was completely re-architected, and part of the management functionality has been moved to kernel mode in order to improve efficiency and performance and minimize complexity. The original thread pool implementation required the user-mode code inside Ntdll.dll to remain aware of how many threads were currently active as worker threads, and to enlarge this number in periods of high demand.

Because querying the information necessary to make this decision, as well as the work to create the threads, took place in user mode, several system calls were required that could have been avoided if these operations were performed in kernel mode. Moving this code into kernel mode means fewer transitions between user and kernel mode, and it allows Ntdll.dll to manage the thread pool itself and not the system mechanisms behind it. It also provides other benefits, such as the ability to remotely create a thread pool in a process other than the calling process (although possible in user mode, it would be very complex given the necessity of using APIs to access the remote process's address space).

The functionality in Windows Vista is introduced by a new object manager type called *TpWorkerFactory,* as well as four new native system calls for managing the factory and its workers—*NtCreateWorkerFactory, NtWorkerFactoryWorkerReady, NtReleaseWorkerFactoryWorker, NtShutdownWorkerFactory*—two new query/set native calls (*NtQueryInformationWorkerFactory* and *NtSetInformationWorkerFactory*), and a new wait call, *NtWaitForWorkViaWorkerFactory.*

Just like other native system calls, these calls provide user mode with a handle to the *TpWorkerFactory* object, which contains information such as the name and object attributes, the desired access mask, and a security descriptor. Unlike other system calls wrapped by the Windows API, however, thread pool management is handled by Ntdll.dll's native code, which means that developers work with an opaque descriptor (a TP_WORK pointer) owned by Ntdll.dll, in which the actual handle is stored.

As its name suggests, the worker factory implementation is responsible for allocating worker threads (and calling the given user-mode worker thread entry point), maintaining a minimum and maximum thread count (allowing for either permanent worker pools or totally dynamic pools), as well as other accounting information. This enables operations such as shutting down the thread pool to be performed with a single call to the kernel, because the kernel has been the only component responsible for thread creation and termination.

Because the kernel dynamically creates new threads as requested, this also increases the scalability of applications using the new thread pool implementation. Developers have always been able to take advantage of as many threads as possible (based on the number of processors on the system) through the old implementation, but through support for dynamic processors in Windows Vista (see the section on this topic later in this chapter), it's now possible for applications using thread pools to automatically take advantage of new processors added at run time.

It's important to note that the new worker factory support is merely a wrapper to manage mundane tasks that would otherwise have to be performed in user mode (at a loss of performance). Many of the improvements in the new thread pool code are the result of changes in the Ntdll.dll side of this architecture. Also, it is not the worker factory code that provides the scalability, wait internals, and efficiency of work processing. Instead, it is a much older component of Windows that we have already discussed—I/O completion ports, or more correctly, kernel queues (KQUEUE; see Chapter 7 for more information).

In fact, when creating a worker factory, an I/O completion port must have already been created by user mode, and the handle needs to be passed on. It is through this I/O completion port that the user-mode implementation will queue work and also wait for work—but by calling the worker factory system calls instead of the I/O completion port APIs. Internally, however, the "release" worker factory call (which queues work) is a wrapper around *IoSetIoCompletion,* which increases pending work, while the "wait" call is a wrapper around *IoRemoveIoCompletion*. Both these routines call into the kernel queue implementation.

Therefore, the job of the worker factory code is to manage either a persistent, static, or dynamic thread pool; wrap the I/O completion port model into interfaces that try to prevent stalled worker queues by automatically creating dynamic threads; and to simplify global cleanup and termination operations during a factory shutdown request (as well as to easily block new requests against the factory in such a scenario).

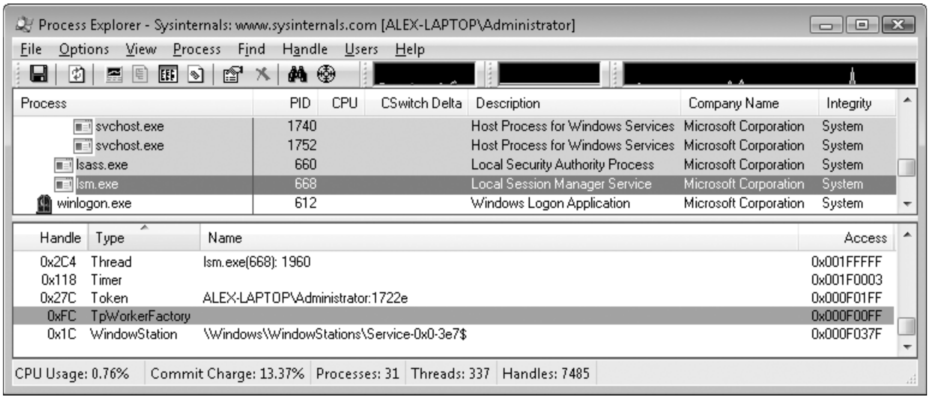
Unfortunately, the data structures used by the worker factory implementation are not in the public symbols, but it is still possible to look at some worker pools, as we'll show in the next experiment.

EXPERIMENT: Looking at Thread Pools

Because of the more efficient and simpler thread pool implementation in Windows Vista, many core system components and applications were updated to make use of it. One of the ways to identify which processes are using a worker factory is to look at the handle list in Process Explorer. Follow these steps to look at some details behind them:

1. Run Process Explorer and select Show Unnamed Handles And Mappings from the View menu. Unfortunately, worker factories aren't named by Ntdll.dll, so you need to take this step in order to see the handles.
2. Select Lsm.exe from the list of processes, and look at the handle table. Make sure that the lower pane is shown (View, Show Lower Pane) and is displaying handle table mode (View, Lower Pane View, Handles).
3. Right-click on the lower pane columns, and then click on Select Columns. Make sure that the Type column is selected to be shown.

4. Now scroll down the handles, looking at the Type column, until you find a handle of type *TpWorkerFactory*. You should see something like this:


Process Explorer - Sysinternals: www.sysinternals.com [ALEX-LAPTOP\Administrator]
File Options View Process Find Handle Users Help

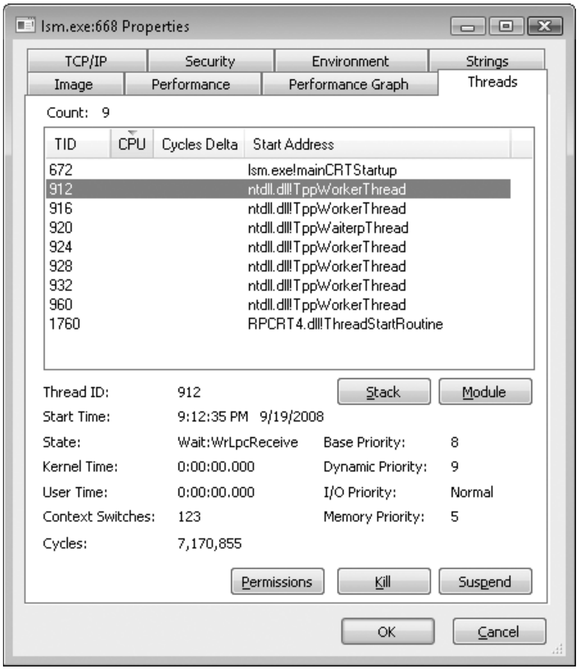
Process	PID	CPU	CSwitch Delta	Description	Company Name	Integrity
svchost.exe	1740			Host Process for Windows Services	Microsoft Corporation	System
svchost.exe	1752			Host Process for Windows Services	Microsoft Corporation	System
lsass.exe	660			Local Security Authority Process	Microsoft Corporation	System
lsm.exe	668			Local Session Manager Service	Microsoft Corporation	System
winlogon.exe	612			Windows Logon Application	Microsoft Corporation	System

Handle	Type	Name	Access
0x2C4	Thread	lsm.exe(668): 1960	0x001FFFFF
0x118	Timer		0x001F0003
0x27C	Token	ALEX-LAPTOP\Administrator:1722e	0x000F01FF
0xFC	TpWorkerFactory		0x000F00FF
0x1C	WindowStation	\Windows\WindowStations\Service-0x0-3e7$	0x000F037F

CPU Usage: 0.76% Commit Charge: 13.37% Processes: 31 Threads: 337 Handles: 7485

Notice how the *TpWorkerFactory* handle is immediately preceded by an *IoCompletion* handle. As was described previously, this occurs because before creating a worker factory, a handle to an I/O completion port on which work will be sent must be created.

5. Now double-click Lsm.exe in the list of processes, and go to the Threads tab. You should see something similar to the image here:



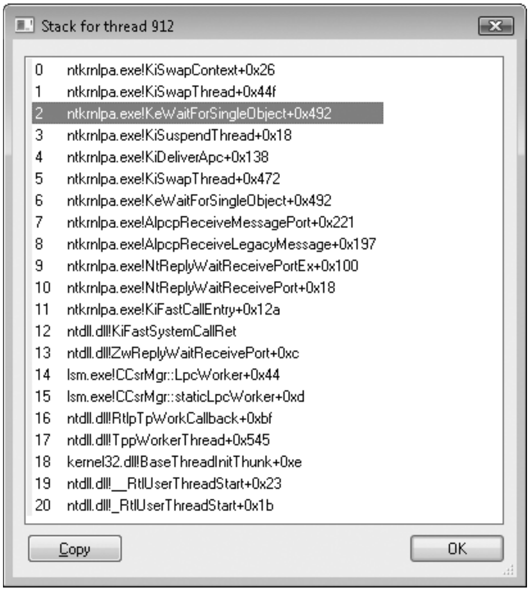
Count: 9

TID	CPU	Cycles Delta	Start Address
672			lsm.exe!mainCRTStartup
912			ntdll.dll!TppWorkerThread
916			ntdll.dll!TppWorkerThread
920			ntdll.dll!TppWaiterpThread
924			ntdll.dll!TppWorkerThread
928			ntdll.dll!TppWorkerThread
932			ntdll.dll!TppWorkerThread
960			ntdll.dll!TppWorkerThread
1760			RPCRT4.dll!ThreadStartRoutine

Thread ID: 912
Start Time: 9:12:35 PM 9/19/2008
State: Wait:WrLpcReceive
Base Priority: 8
Kernel Time: 0:00:00.000
Dynamic Priority: 9
User Time: 0:00:00.000
I/O Priority: Normal
Context Switches: 123
Memory Priority: 5
Cycles: 7,170,855

On this system (with two processors), the worker factory has created six worker threads at the request of Lsm.exe (processes can define a minimum and maximum number of threads) and based on its usage and the count of processors on the machine. These threads are identified as *TppWorkerThread*, which is Ntdll.dll's worker entry point when calling the worker factory system calls.

6. Ntdll.dll is responsible for its own internal accounting inside the worker thread wrapper (*TppWorkerThread)* before calling the worker callback that the application has registered. By looking at the Wait reason in the State information for each thread, you can get a rough idea of what each worker thread may be doing. Double-click on one of the threads inside an LPC wait to look at its stack. Here's an example:



```
0    ntkrnlpa.exe!KiSwapContext+0x26
1    ntkrnlpa.exe!KiSwapThread+0x44f
2    ntkrnlpa.exe!KeWaitForSingleObject+0x492
3    ntkrnlpa.exe!KiSuspendThread+0x18
4    ntkrnlpa.exe!KiDeliverApc+0x138
5    ntkrnlpa.exe!KiSwapThread+0x472
6    ntkrnlpa.exe!KeWaitForSingleObject+0x492
7    ntkrnlpa.exe!AlpcpReceiveMessagePort+0x221
8    ntkrnlpa.exe!AlpcpReceiveLegacyMessage+0x197
9    ntkrnlpa.exe!NtReplyWaitReceivePortEx+0x100
10   ntkrnlpa.exe!NtReplyWaitReceivePort+0x18
11   ntkrnlpa.exe!KiFastCallEntry+0x12a
12   ntdll.dll!KiFastSystemCallRet
13   ntdll.dll!ZwReplyWaitReceivePort+0xc
14   lsm.exe!CCsrMgr::LpcWorker+0x44
15   lsm.exe!CCsrMgr::staticLpcWorker+0xd
16   ntdll.dll!RtlpTpWorkCallback+0xbf
17   ntdll.dll!TppWorkerThread+0x545
18   kernel32.dll!BaseThreadInitThunk+0xe
19   ntdll.dll!__RtlUserThreadStart+0x23
20   ntdll.dll!_RtlUserThreadStart+0x1b
```

This specific worker thread is being used by Lsm.exe for LPC communication. Because the local session manager needs to communicate with other components such as Smss and Csrss through LPC, it makes sense that it would want a number of its threads to be busy replying and waiting for LPC messages (the more threads doing this, the less stalling on the LPC pipeline).

If you look at other worker threads, you'll see some are waiting for objects such as events. A process can have multiple thread pools, and each thread pool can have a variety of threads doing completely unrelated tasks. It's up to the developer to assign work and to call the thread pool APIs to register this work through Ntdll.dll.

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 functions and Windows utilities and tools that relate to scheduling, the detailed data structures and algorithms that make up the Windows scheduling system are presented, with uniprocessor systems examined first and then multiprocessor systems.

Overview of Windows Scheduling

Windows implements a priority-driven, preemptive scheduling system—the highest-priority runnable (*ready*) thread always runs, with the caveat that the thread chosen to run might be limited by the processors on which the thread is allowed to run, a phenomenon called *processor affinity*. By default, threads can run on any available processor, but you can alter processor affinity by using one of the Windows scheduling functions listed in Table 5-15 (shown later in the chapter) or by setting an affinity mask in the image header.

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, five threads are ready to run at priority 8 on the first processor, and three threads at priority 10, two threads at priority 9, and six threads at priority 8 are ready to run on the second processor. Determining which of these threads get to run on their respective processor 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
    THREAD 8576c030  Cid 0ec8.0c9c  Teb: 7ffa8000 Win32Thread: 00000000 READY
    THREAD 85a8a7f0  Cid 0ec8.0d3c  Teb: 7ff97000 Win32Thread: 00000000 READY
    THREAD 87d34488  Cid 0c48.04a0  Teb: 7ffde000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 10
    THREAD 857c0030  Cid 04c8.0378  Teb: 7ffdf000 Win32Thread: fef7f8c0 READY
    THREAD 856cc8e8  Cid 0e84.0a70  Teb: 7ffdb000 Win32Thread: f98fb4c0 READY
    THREAD 85c41c68  Cid 0e84.00ac  Teb: 7ffde000 Win32Thread: ff460668 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
    THREAD 86086278  Cid 0ec8.0cc8  Teb: 7ff8d000 Win32Thread: 00000000 READY
    THREAD 8816f7f0  Cid 0ec8.0b60  Teb: 7ffd5000 Win32Thread: 00000000 READY
    THREAD 87710d78  Cid 0004.01b4  Teb: 00000000 Win32Thread: 00000000 READY
```

When 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 (or higher, which can occur on a multiprocessor system) 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), foreground/background status of the process, or use of the job object to alter the quantum. (Quantums are described in more detail in the "Quantum" section later in the chapter.) 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. When Windows selects a new thread to run, it performs a *context switch* to it. A context switch is the procedure of saving the volatile machine 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, you must first understand the priority levels that Windows uses. As illustrated in Figure 5-12, internally Windows uses 32 priority levels, ranging from 0 through 31. These values divide up as follows:

- Sixteen real-time levels (16 through 31)
- Fifteen variable levels (1 through 15)
- One system level (0), reserved for the zero page thread

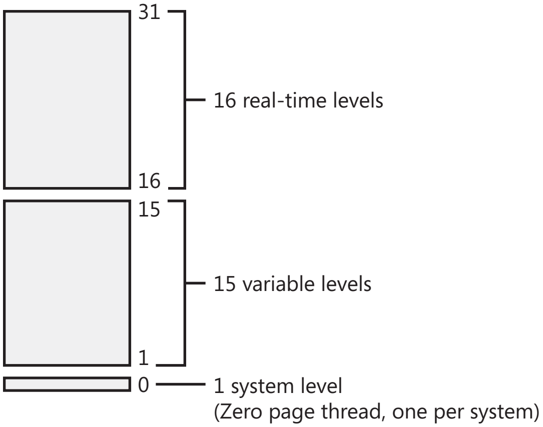


FIGURE 5-12 Thread priority levels

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 (Real-time, High, Above Normal, Normal, Below Normal, and Idle) and then by the relative priority of the individual threads within those processes (Time-critical, Highest, Above-normal, Normal, Below-normal, Lowest, and Idle).

In the Windows API, each thread has a base priority that is a function of its process priority class and its relative thread priority. The mapping from Windows priority to internal Windows numeric priority is shown in Figure 5-13.

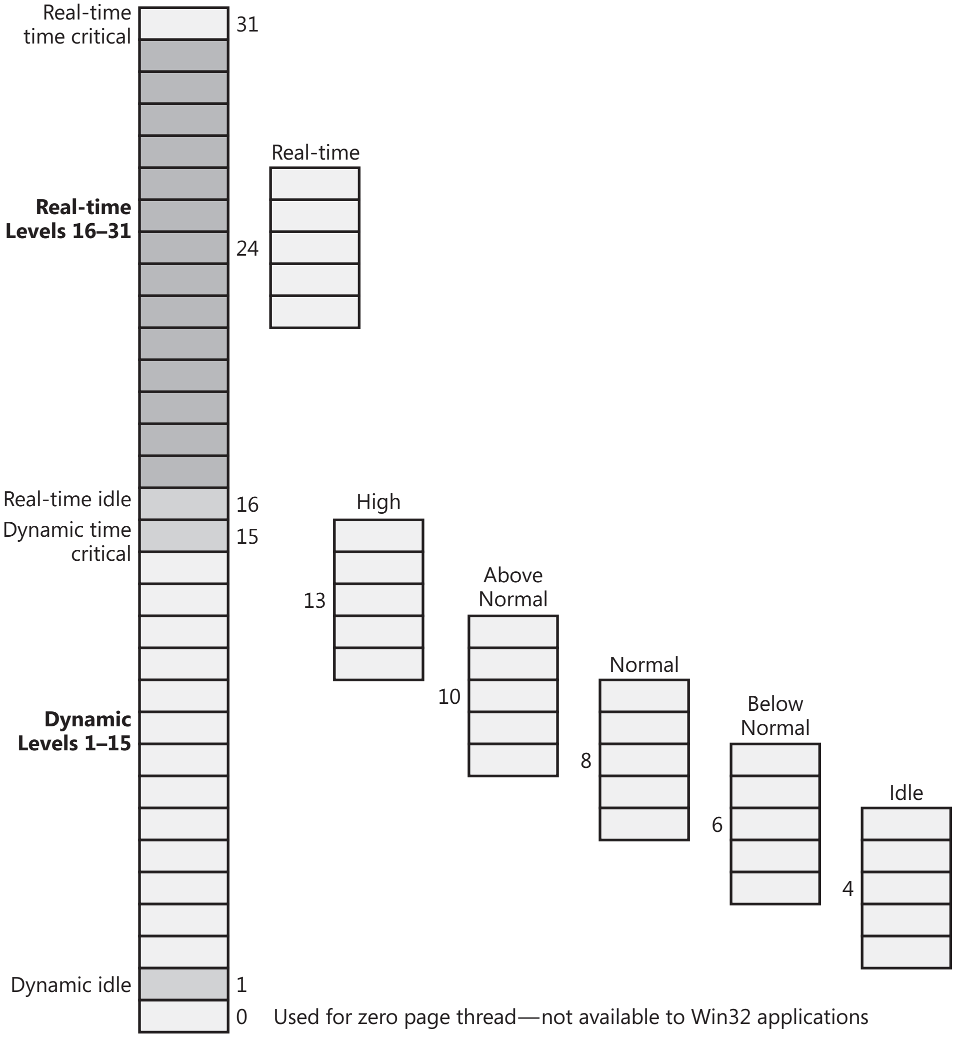


FIGURE 5-13 Mapping of Windows kernel priorities to the Windows API

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 (1 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. It usually doesn't make sense, however, to change individual thread priorities within a process, because unless you wrote the program or have the source code, you don't really know what the individual threads are doing, and changing their relative importance might cause the program not to behave in the intended fashion.

Normally, the process base priority (and therefore the starting thread base priority) will default to the value at the middle of each process priority range (24, 13, 10, 8, 6, or 4). However, some Windows system processes (such as the Session Manager, service controller, and local security authentication server) 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. These system processes use an internal system call (*NtSetInformationProcess*) to set their process base priority to a numeric value other than the normal default starting base priority.

Windows Scheduling APIs

The Windows API functions that relate to thread scheduling are listed in Table 5-15. (For more information, see the Windows API reference documentation.)

TABLE 5-15 Scheduling-Related APIs and Their Functions

API	Function
Suspend/ResumeThread	Suspends or resumes a paused thread from execution.
Get/SetPriorityClass	Returns or sets a process's priority class (base priority).
Get/SetThreadPriority	Returns or sets a thread's priority (relative to its process base priority).
Get/SetProcessAffinityMask	Returns or sets a process's affinity mask.
SetThreadAffinityMask	Sets a thread's affinity mask (must be a subset of the process's affinity mask) for a particular set of processors, restricting it to running on those processors.
SetInformationJobObject	Sets attributes for a job; some of the attributes affect scheduling, such as affinity and priority. (See the "Job Objects" section later in the chapter for a description of the job object.)
GetLogicalProcessorInformation	Returns details about processor hardware configuration (for hyperthreaded and NUMA systems).
Get/SetThreadPriorityBoost	Returns or sets the ability for Windows to boost the priority of a thread temporarily. (This ability applies only to threads in the dynamic range.)
SetThreadIdealProcessor	Establishes a preferred processor for a particular thread, but doesn't restrict the thread to that processor.

API	Function
Get/SetProcessPriorityBoost	Returns or sets the default priority boost control state of the current process. (This function is used to set the thread priority boost control state when a thread is created.)
WaitForSingle/MultipleObject(s)	Puts the current thread into a wait state until the specified object(s) is/are satisfied, or until the specified time interval (figured in milliseconds [msec]) expires, if given.
SwitchToThread	Yields execution to another thread (at priority 1 or higher) that is ready to run on the current processor.
Sleep	Puts the current thread into a wait state for a specified time interval (figured in milliseconds [msec]). A zero value relinquishes the rest of the thread's quantum.
SleepEx	Causes the current thread to go into a wait state until either an I/O completion callback is completed, an APC is queued to the thread, or the specified time interval ends.

Relevant Tools

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).

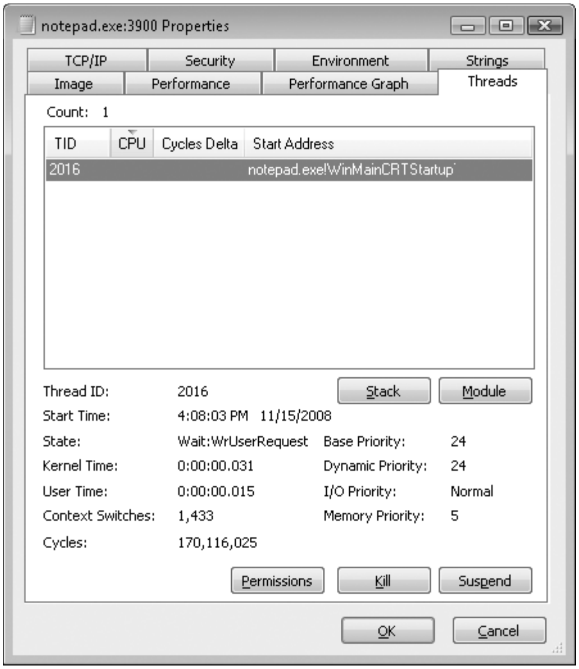
You can view individual thread priorities with the Reliability and Performance Monitor, Process Explorer, or WinDbg. While it might be useful to increase or lower the priority of a process, it typically does not make sense to adjust individual thread priorities within a process because only a person 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 would be **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 this image:



Count: 1

TID	CPU	Cycles Delta	Start Address
2016			notepad.exe!WinMainCRTStartup

Thread ID:	2016		
Start Time:	4:08:03 PM 11/15/2008		
State:	Wait:WrUserRequest	Base Priority:	24
Kernel Time:	0:00:00.031	Dynamic Priority:	24
User Time:	0:00:00.015	I/O Priority:	Normal
Context Switches:	1,433	Memory Priority:	5
Cycles:	170,116,025		

3. Task Manager can show you similar information. Press Ctrl+Shift+Esc to start Task Manager, and go to 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.

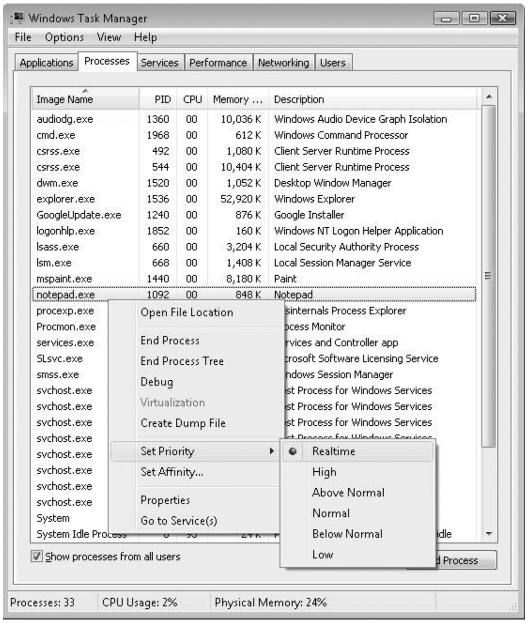


Image Name	PID	CPU	Memory ...	Description
audiodg.exe	1360	00	10,036 K	Windows Audio Device Graph Isolation
cmd.exe	1968	00	612 K	Windows Command Processor
csrss.exe	492	00	1,080 K	Client Server Runtime Process
csrss.exe	544	00	10,404 K	Client Server Runtime Process
dwm.exe	1520	00	1,052 K	Desktop Window 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	0	95	24 K	idle

☑ Show processes from all users

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

Windows System Resource Manager

Windows Server 2008 Enterprise Edition and Windows Server 2008 Datacenter Edition 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 have 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 9 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 could be used to detect a process with a memory leak before it consumes all the available committed virtual 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).

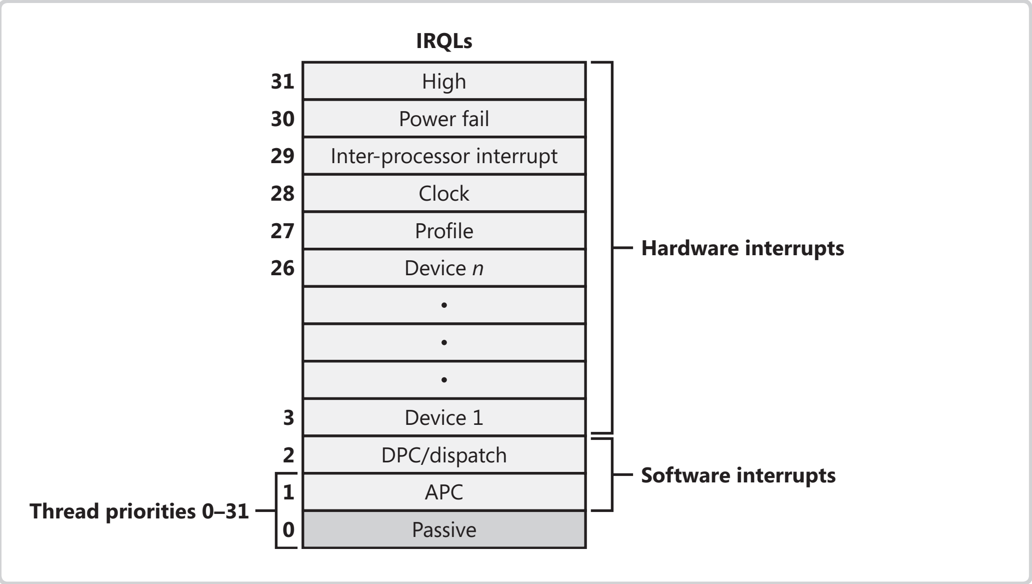
Real-Time Priorities

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).

Note As illustrated in the following figure showing the x86 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. For more information, see the sidebar "Windows and Real-Time Processing" in Chapter 3 as well as the MSDN Library article "Real-Time Systems and Microsoft Windows NT."

Interrupt Levels vs. Priority Levels

As illustrated in the following figure of the interrupt request levels (IRQLs) for a 32-bit system, threads normally run at IRQL 0 or 1. (For a description of how Windows uses interrupt levels, see Chapter 3.) User-mode code always runs at IRQL 0. Because of this, no user-mode thread, regardless of its priority, blocks hardware interrupts (although high-priority real-time threads can block the execution of important system threads). Only kernel-mode APCs execute at IRQL 1 because they interrupt the execution of a thread. (For more information on APCs, see Chapter 3.) Threads running in kernel mode can raise IRQL to higher levels, though—for example, while executing a system call that involves thread dispatching.



Thread States

Before you can comprehend the thread-scheduling algorithms, you need to understand the various execution states that a thread can be in. Figure 5-14 illustrates the state transitions for threads. (The numeric values shown represent the value of the thread state performance counter.) More details on what happens at each transition are included later in this section.

The thread states are as follows:

- **Ready** A thread in the ready state is waiting to execute. 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 yet been scheduled. This state exists so that the kernel can minimize the amount of time the systemwide 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 wait state.

- **Waiting** A thread can enter the wait 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.
- **Gate Waiting** When a thread does a wait on a gate dispatcher object (see Chapter 3 for more information on gates), it enters the gate waiting state instead of the waiting state. This difference is important when breaking a thread's wait as the result of an APC. Because gates don't use the dispatcher lock, but a per-object lock, the kernel needs to perform some unique locking operations when breaking the wait of a thread waiting on a gate and a way to differentiate this from a normal wait.
- **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 block (the data structure in nonpaged pool that describes the thread) might or might not be deallocated. (The object manager sets policy regarding when to delete the object.)
- **Initialized** This state is used internally while a thread is being created.

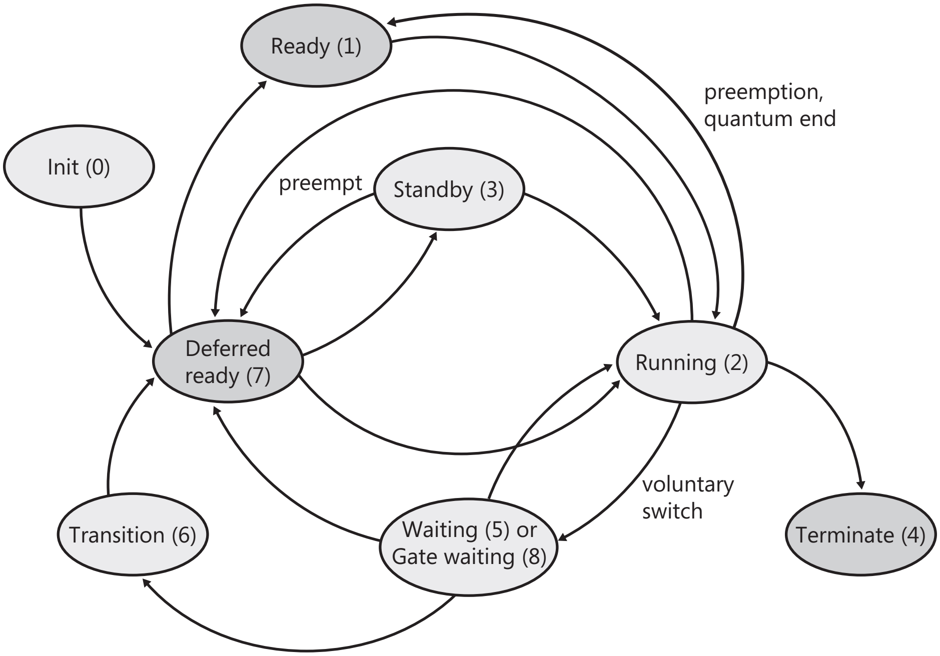
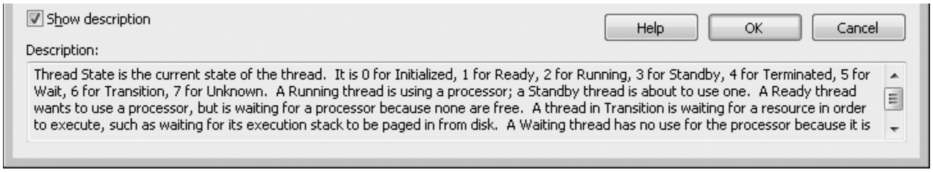


FIGURE 5-14 Thread states and transitions

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:

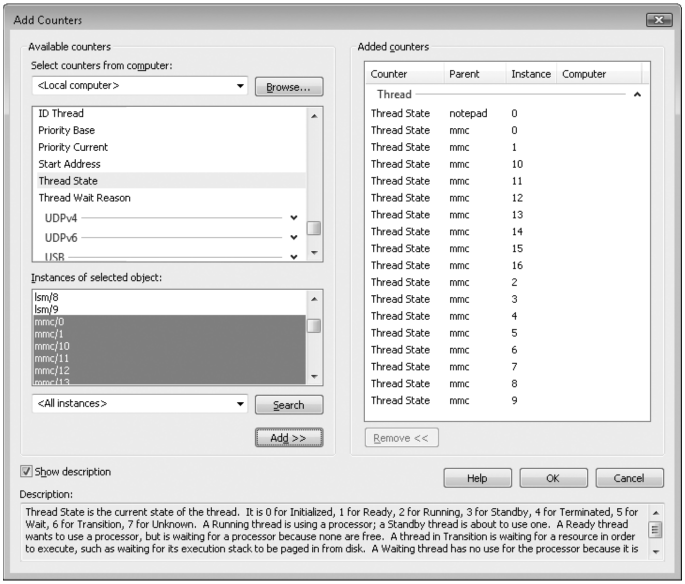
1. Run Notepad (Notepad.exe).
2. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability and Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
3. Select chart view if you're in some other view.
4. Right-click on the graph, and choose Properties.
5. Click 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:



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 click 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 following dialog box.



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	

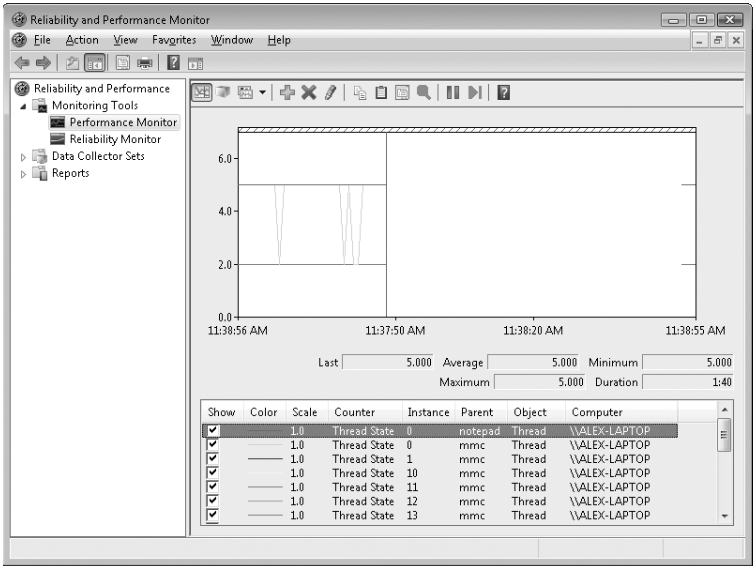
☑ Show description

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, which, as shown in the explanation text you saw under step 7, 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 5-15. 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 5-15. 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. (Versions of Windows before Windows Server 2003 used a global database).

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!_prcb** 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 may sometimes see the Windows scheduling algorithm referred to as an O(1), or constant time, algorithm.

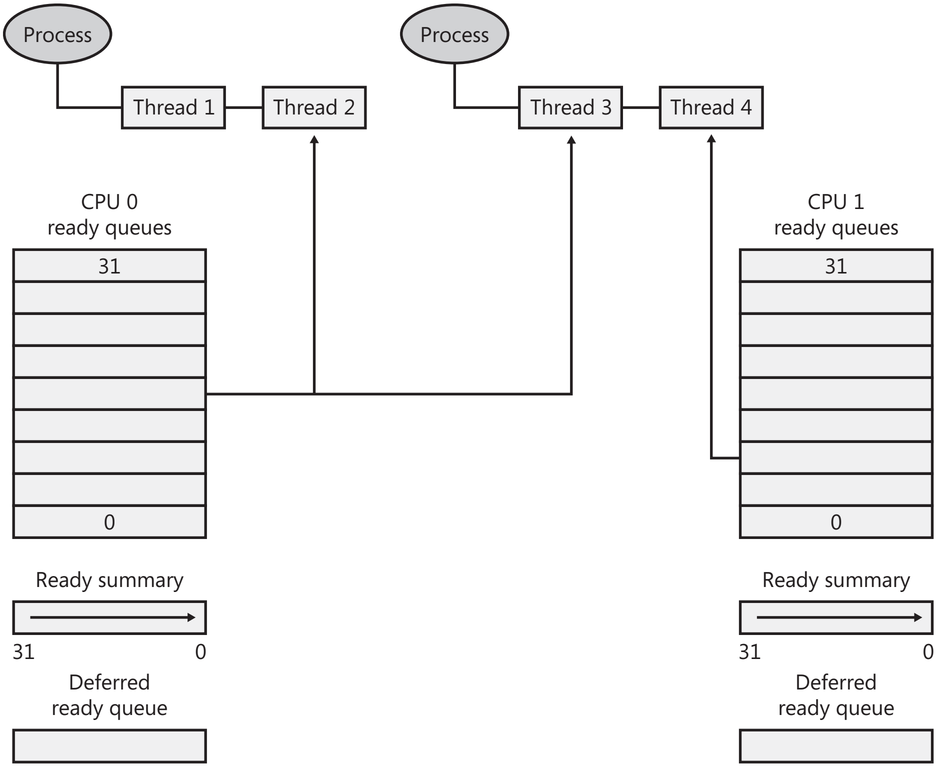


FIGURE 5-15 Windows multiprocessor dispatcher database

Table 5-16 lists the KPRCB fields involved in thread scheduling.

TABLE 5-16 Thread-Scheduling KPRCB Fields

KPRCB Field	Type	Description
ReadySummary	Bitmask (32 bits)	Bitmask of priority levels that have one or more ready threads
DeferredReadyListHead	Singly linked list	Single list head for the deferred ready queue
DispatcherReadyListHead	Array of 32 list entries	List heads for the 32 ready queues

The dispatcher database is synchronized by raising IRQL to SYNCH_LEVEL (which is defined as level 2). (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, on a multiprocessor system, more is required than just raising IRQL because other processors can simultaneously raise to the same IRQL and attempt to operate on the 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 Windows Vista, threads run by default for 2 clock intervals; on Windows 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, 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 of changes in thread run-time accounting in Windows Vista (briefly mentioned earlier in the thread activity experiment), 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 above).

The end result of this new accounting method is that, as of Windows Vista, 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 we've just seen, 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 may 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 32-bit Windows Vista system:

```
C:\>clockres

ClockRes - View the system clock resolution
By Mark Russinovich
SysInternals - www.sysinternals.com

The system clock interval is 15.600100 ms
```

Quantum Accounting

Each process has a quantum reset value in the kernel process block. This value is used when creating new threads inside the process and is duplicated in the kernel thread block, 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, then 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 Windows Vista systems, threads, by default, have a quantum reset value of 6 (2 * 3), and that Windows Server 2008 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, since the original value would describe 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 second.
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 hexidecimal. 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.

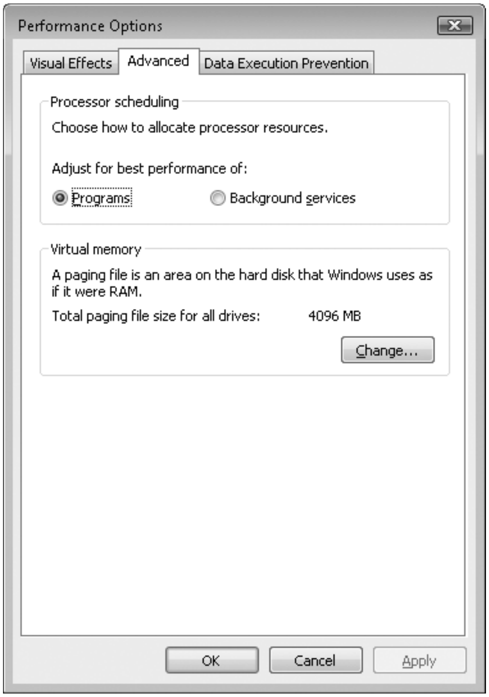
```
lkd> dd nt!KiCyclesPerClockQuantum l1
81d31ae8  00e0786e
```

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, the default for client machines) or long (12 clock ticks, 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 name's icon on the desktop, choose Properties, click the Advanced System Settings label, select the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. The dialog box displayed is shown in Figure 5-16.



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 5-16 Quantum configuration in the Performance Options dialog box

The Programs setting designates the use of short, variable quantums—the default for Windows Vista. If you install Terminal Services on Windows Server 2008 systems and configure the server as an application server, this setting is selected so that the users on the terminal server will 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 Windows Server 2008 systems. The only reason you might select this option on a workstation system is if you were using the workstation as a server system.

One additional difference between the Programs and Background Services settings is the effect they have on the quantum of the threads in the foreground process. This is explained in the next section.

Quantum Boosting

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. Thus, 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).

Note that this adjustment of quantums applies only to processes with a priority higher than Idle on systems configured to Programs in the Performance Options settings described in the previous section. Thread quantums are not changed for the foreground process on systems configured to Background Services (the default on Windows Server 2008 systems).

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 threads in the foreground process should have their quantums boosted (and if so, the amount of the boost). This value consists of 6 bits divided into the three 2-bit fields shown in Figure 5-17.

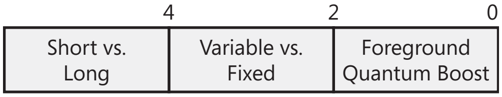


FIGURE 5-17 Fields of the Win32PrioritySeparation registry value

The fields shown in Figure 5-17 can be defined as follows:

- **Short vs. Long** A setting of 1 specifies long, and 2 specifies short. A setting of 0 or 3 indicates that the default will be used (short for Windows Vista, long for Windows Server 2008 systems).
- **Variable vs. Fixed** A setting of 1 means to vary the quantum for the foreground process, and 2 means that quantum values don't change for foreground processes. A setting of 0 or 3 means that the default (which is variable for Windows Vista and fixed for Windows Server 2008 systems) will be used.

- **Foreground Quantum Boost** This field (stored in the kernel variable *PsPrioritySeperation*) must have a value of 0, 1, or 2. (A setting of 3 is invalid and treated as 2.) It is used as an index into a three-element byte array named *PspForegroundQuantum* to obtain the quantum for the threads in the foreground process. The quantum for threads in background processes is taken from the first entry in this quantum table. Table 5-17 shows the possible settings for *PspForegroundQuantum*.

TABLE 5-17 Quantum Values

	Short			Long		
Variable	6	12	18	12	24	36
Fixed	18	18	18	36	36	36

Note that when you're using the Performance Options dialog box described earlier, 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.

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 *PsPrioritySeperation* 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, select the Advanced tab, click the Settings button in the Performance section, and finally click 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 *PsPrioritySeperation* (this is a deliberate misspelling inside the Windows kernel, not an error in this book) and *PspForegroundQuantum,* as shown here. The values shown are what you should see on a Windows Vista 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 PsPrioritySeperation 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, since 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 _KPROCESS 85b32d90
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY [ 0x85b32da0 - 0x85b32da0 ]
   +0x018 DirectoryTableBase : 0xb45b0880
   +0x01c Unused0          : 0
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : 0x20ac
   +0x032 Iopl             : 0 ''
   +0x033 Unused           : 0 ''
   +0x034 ActiveProcessors : 1
   +0x038 KernelTime       : 0
   +0x03c UserTime         : 0
   +0x040 ReadyListHead    : _LIST_ENTRY [ 0x85b32dd0 - 0x85b32dd0 ]
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : (null)
   +0x050 ThreadListHead   : _LIST_ENTRY [ 0x861e7e0c - 0x8620637c ]
   +0x058 ProcessLock      : 0
   +0x05c Affinity         : 3
   +0x060 AutoAlignment    : 0y0
   +0x060 DisableBoost     : 0y0
   +0x060 DisableQuantum   : 0y0
   +0x060 ReservedFlags    : 0y00000000000000000000000000000 (0)
   +0x060 ProcessFlags     : 0
   +0x064 BasePriority     : 8 ''
   +0x065 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 PsPrioritySeperation L1
81d3101c  00000000
lkd> db PspForegroundQuantum l 3
81d0946c  24 24 24                                         $$$
lkd> dt _KPROCESS 85b32d90
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY [ 0x85b32da0 - 0x85b32da0 ]
   +0x018 DirectoryTableBase : 0xb45b0880
   +0x01c Unused0          : 0
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : 0x20ac
   +0x032 Iopl             : 0 ''
   +0x033 Unused           : 0 ''
   +0x034 ActiveProcessors : 1
   +0x038 KernelTime       : 0
   +0x03c UserTime         : 0
   +0x040 ReadyListHead    : _LIST_ENTRY [ 0x85b32dd0 - 0x85b32dd0 ]
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : (null)
   +0x050 ThreadListHead   : _LIST_ENTRY [ 0x861e7e0c - 0x860c14f4 ]
   +0x058 ProcessLock      : 0
   +0x05c Affinity         : 3
   +0x060 AutoAlignment    : 0y0
   +0x060 DisableBoost     : 0y0
   +0x060 DisableQuantum   : 0y0
   +0x060 ReservedFlags    : 0y00000000000000000000000000000 (0)
   +0x060 ProcessFlags     : 0
   +0x064 BasePriority     : 8 ''
   +0x065 QuantumReset     : 36 '$'
```

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 5-18 illustrates a thread entering a wait state and Windows selecting a new thread to run.

In Figure 5-18, 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.

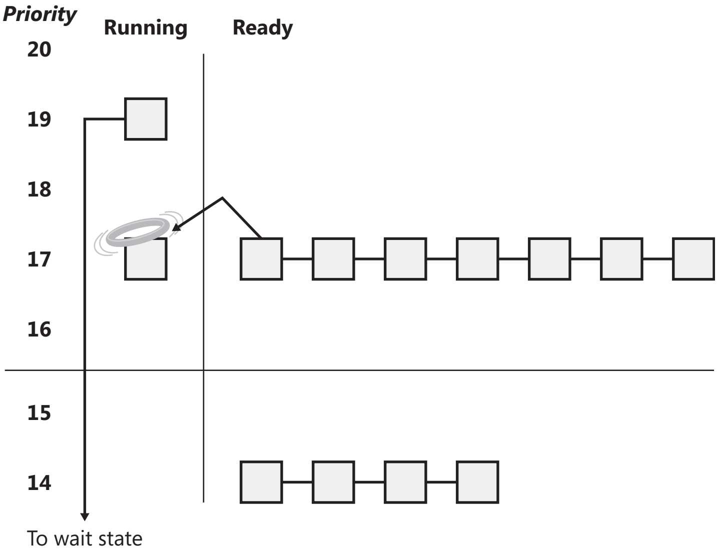


FIGURE 5-18 Voluntary switching

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 5-19 illustrates this situation.

In Figure 5-19, 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.

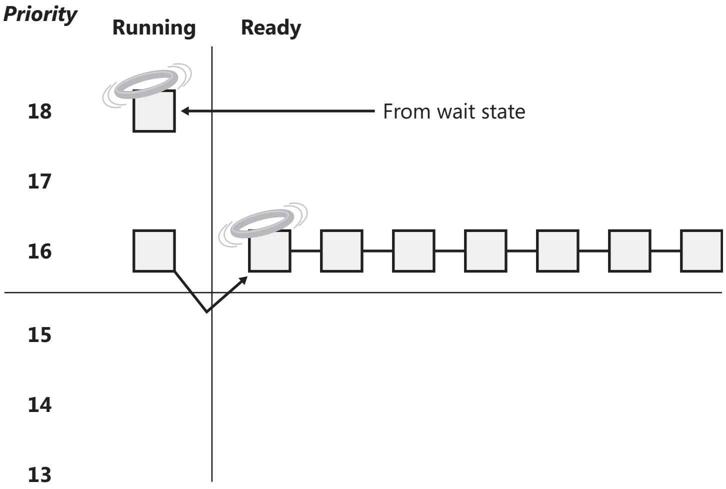


FIGURE 5-19 Preemptive thread scheduling

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 5-20. If no other thread of the same priority is ready to run, the thread gets to run for another quantum.

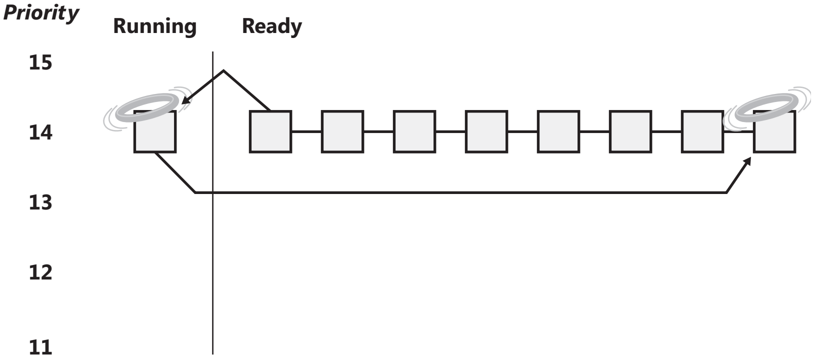


FIGURE 5-20 Quantum end thread scheduling

As we've seen, 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 may have happened previously and is important to discuss.

Under the scheduling model prior to Windows Vista, which relied only on the clock interval timer, the following situation could 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, thread *A* starts running again, but it quickly hits the next clock interval. The scheduler can only assume 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 preemption or interrupt handling).

In this scenario, thread *A* was unfairly penalized in two different ways. First of all, the time that it had to spend handling a device interrupt was accounted to its own CPU time, even though the thread had probably nothing to do with the interrupt. (Recall that interrupts are handled in the context of whichever thread had been 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 5-21 represents this scenario.

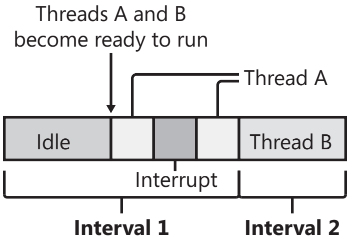


FIGURE 5-21 Unfair time slicing in previous versions of Windows

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 will occur:

- 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, thread *A* starts running again, but it quickly hits the next clock interval. The scheduler looks at the number of CPU clock cycles that have been 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 may have additionally been 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 5-22 represents this scenario.

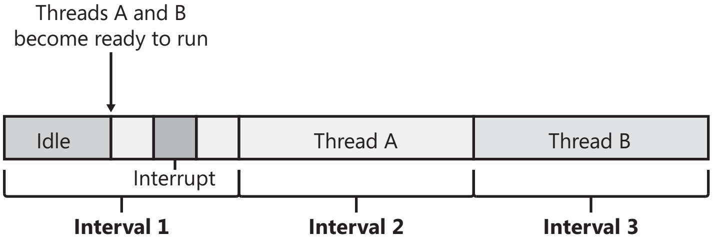


FIGURE 5-22 Fair time slicing in current versions of Windows

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.

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's 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 block. 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 9.) If a kernel APC that needs to be delivered is pending, an interrupt at IRQL 1 is requested. Otherwise, control passes to the new thread's restored instruction pointer and the new thread resumes execution.

Idle Thread

When no runnable thread exists on a CPU, Windows dispatches the per-CPU idle thread. Each CPU is allotted one idle thread because on a multiprocessor system one CPU can be executing a thread while other CPUs might have no threads to execute.

Various Windows process viewer utilities report the idle process using different names. Task Manager and Process Explorer call it "System Idle Process," while Tlist calls it "System Process." If you look at the EPROCESS structure's *ImageFileName* member, you'll see the internal name for the process is "Idle." Windows reports the priority of the idle thread as 0 (15 on x64 systems). In reality, however, the idle threads don't have a priority level because they run only when there are no real threads to run—they are not scheduled and 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 9.)

Apart from priority, there are many other fields in the idle process or its threads that may be reported as 0. This occurs because the idle process is not an actual full-blown object manager process object, and neither are its idle threads. Instead, the initial idle thread and idle process objects are statically allocated and used to bootstrap the system before the process manager initializes. Subsequent idle thread structures are allocated dynamically as additional processors are brought online. Once process management initializes, it uses the special variable *PsIdleProcess* to refer to the idle process.

Apart from some critical fields provided so that these threads and their process can have a PID and name, everything else is ignored, which means that query APIs may simply return zeroed data.

The idle loop runs at DPC/dispatch level, polling for work to do, such as delivering deferred procedure calls (DPCs) or looking for threads to dispatch to. Although some details of the flow vary between architectures, the basic flow of control of the idle thread is as follows:

1. Enables and disables interrupts (allowing any pending interrupts to be delivered).
2. Checks whether any DPCs (described in Chapter 3) are pending on the processor. If DPCs are pending, clears the pending software interrupt and delivers them. (This will also perform timer expiration, as well as deferred ready processing. The latter is explained in the upcoming multiprocessor scheduling section.)
3. Checks whether a thread has been selected to run next on the processor, and if so, dispatches that thread.
4. 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.
5. On debug systems, checks if there is a kernel debugger trying to break into the system and gives it access.
6. If requested, checks for threads waiting to run on other processors and schedules them locally. (This operation is also explained in the upcoming multiprocessor scheduling section.)

Priority Boosts

In six cases, the Windows scheduler can boost (increase) the current priority value of threads:

- On completion of I/O operations
- After waiting for executive events or semaphores
- When a thread has been waiting on an executive resource for too long
- After threads in the foreground process complete a wait operation
- When GUI threads wake up because of windowing activity
- When a thread that's ready to run hasn't been running for some time (CPU starvation)

The intent of these adjustments is to improve overall system throughput and responsiveness as well as resolve potentially unfair scheduling scenarios. 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.

Windows Vista adds one more scenario in which a priority boost can occur, multimedia playback. Unlike the other priority boosts, which are applied directly by kernel code, multimedia playback boosts are managed by a user-mode service called the MultiMedia Class Scheduler Service (MMCSS). (Although the boosts are still done *in* kernel mode, the request to boost the threads is managed by this user-mode service.) We'll first cover the typical kernel-managed priority boosts and then talk about MMCSS and the kind of boosting it performs.

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 will have more of a chance to run right away and process whatever was being waited for. Recall that 1 quantum unit is deducted from the thread's remaining quantum when it wakes up so that I/O bound threads aren't unfairly favored. 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 5-18.) 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 5-18, notice that I/O requests to devices that warrant better responsiveness have higher boost values.

TABLE 5-18 Recommended Boost Values

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

The boost is always applied to a thread's current priority, not its base priority. As illustrated in Figure 5-23, after the boost is applied, the thread gets to run for one quantum at the elevated priority level. After the thread has completed its quantum, it decays one priority level and then runs another quantum. This cycle continues until the thread's priority level has decayed back to its base priority. A thread with a higher priority can still preempt the boosted thread, but the interrupted thread gets to finish its time slice at the boosted priority level before it decays to the next lower priority.

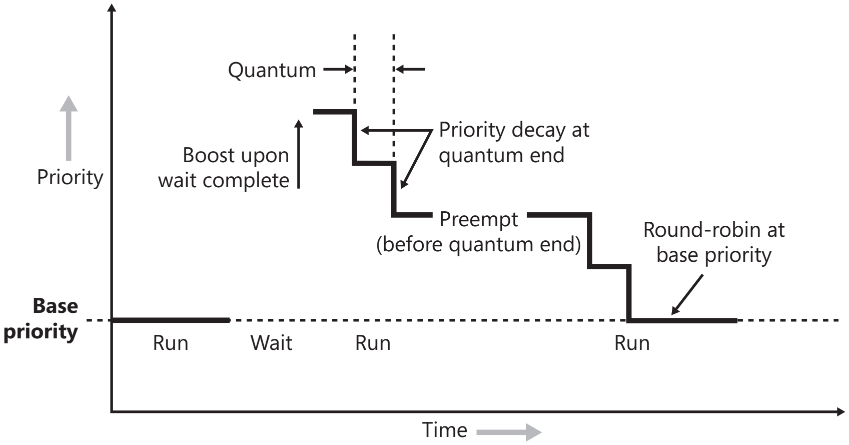

Quantum
Priority
Boost upon wait complete
Priority decay at quantum end
Preempt (before quantum end)
Round-robin at base priority
Base priority
Run
Wait
Run
Run
Time

FIGURE 5-23 Priority boosting and decay

As noted earlier, these boosts apply only to threads in the dynamic priority range (0 through 15). No matter how large the boost is, the thread will never be boosted beyond level 15 into the real-time priority range. In other words, a priority 14 thread that receives a boost of 5 will go up to priority 15. A priority 15 thread that receives a boost will remain at priority 15.

Boosts After Waiting for Events and Semaphores

When a thread that was waiting for an executive event or a semaphore object has its wait satisfied (because of a call to the function *SetEvent*, *PulseEvent*, or *ReleaseSemaphore*), it receives a boost of 1. (See the value for EVENT_ INCREMENT and SEMAPHORE_INCREMENT in the WDK header files.) Threads that wait for events and semaphores warrant a boost for the same reason that threads that wait for I/O operations do—threads that block on events are requesting CPU cycles less frequently than CPU-bound threads. This adjustment helps balance the scales.

This boost operates the same as the boost that occurs after I/O completion, as described in the previous section:

- The boost is always applied to the base priority (not the current priority).
- The priority will never be boosted above 15.
- The thread gets to run at the elevated priority for its remaining quantum (as described earlier, quantums are reduced by 1 when threads exit a wait) before decaying one priority level at a time until it reaches its original base priority.

A special boost is applied to threads that are awoken as a result of setting an event with the special functions *NtSetEventBoostPriority* (used in Ntdll.dll for critical sections) and *KeSetEventBoostPriority* (used for executive resources) or if a signaling gate is used (such as with pushlocks). If a thread waiting for an event is woken up as a result of the special event boost function and its priority is 13 or below, it will have its priority boosted to be the setting thread's priority plus one. If its quantum is less than 4 quantum units, it is set to 4 quantum units. This boost is removed at quantum end.

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 avoid 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 will attempt to prevent CPU starvation by acquiring the dispatcher lock, boosting the owning thread or threads, and performing another wait. Because the dispatcher lock is held and the thread's *WaitNext* flag is set to TRUE, this ensures a consistent state during the boosting process until the next wait is done.

This boost operates in the following manner:

- The boost is always applied to the base priority (not the current priority) of the owner thread.
- The boost raises priority to 14.
- The boost is only applied if the owner thread has a lower priority than the waiting thread, and only if the owner thread's priority isn't already 14.
- The quantum of the thread is reset so that the thread gets to run at the elevated priority for a full quantum, instead of only the quantum it had left. Just like other boosts, at each quantum end, the priority boost will slowly decrease by one level.

Because executive resources can be either shared or exclusive, the kernel will first boost the exclusive owner and then check for shared owners and boost 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. It's important to 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 will boost all those threads to priority 14, resulting in a sudden surge of high-priority threads on the system, all with full quantums. Although the exclusive thread will run first (since it was the first to be boosted and therefore first on the ready list), the other shared owners will run next, since the waiting thread's priority was not boosted. Only until after all the shared owners have gotten a chance to run and their priority 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

Whenever a thread in the foreground process completes a wait operation on a kernel object, the kernel function *KiUnwaitThread* boosts its current (not base) priority by the current value of *PsPrioritySeperation*. (The windowing system is responsible for determining which process is considered to be in the foreground.) As described in the section on quantum controls, *PsPrioritySeperation* 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.

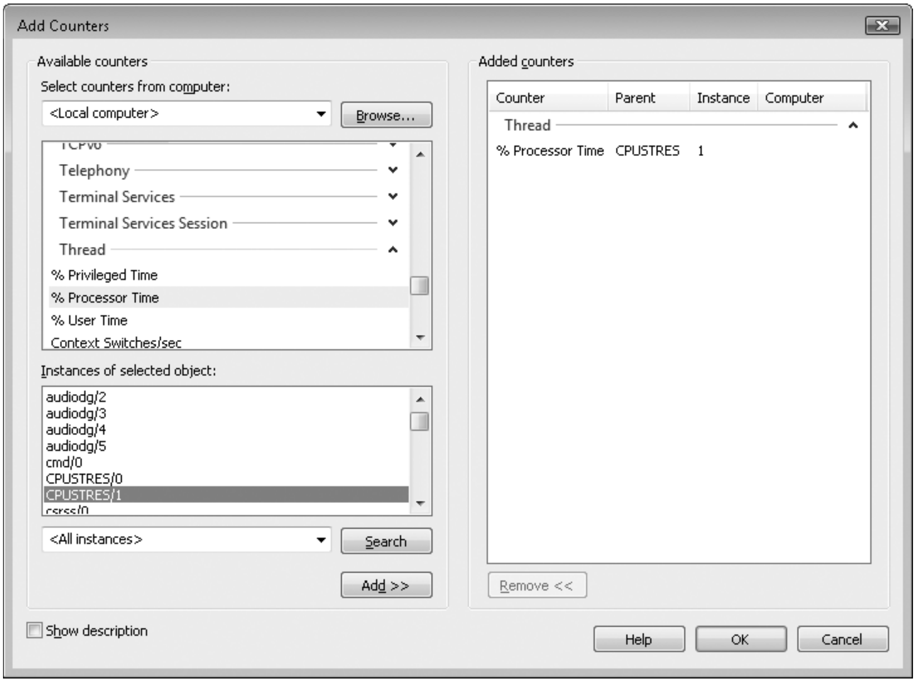
Unlike other types of boosting, this boost applies to all Windows systems, and you *can't* disable this boost, even if you've disabled priority boosting using the Windows *SetThreadPriorityBoost* function.

EXPERIMENT: Watching Foreground Priority Boosts and Decays

Using the CPU Stress tool, 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, select the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. Select the Programs option. This causes *PsPrioritySeperation* 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 Reliability And 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:


Add Counters

Available counters

Select counters from computer: <Local computer>

TCPv6
Telephony
Terminal Services
Terminal Services Session
Thread
% Privileged Time
% Processor Time
% User Time
Context Switches/sec

Instances of selected object:

audiodg/2
audiodg/3
audiodg/4
audiodg/5
cmd/0
CPUSTRES/0
CPUSTRES/1
csrss/0

<All instances>

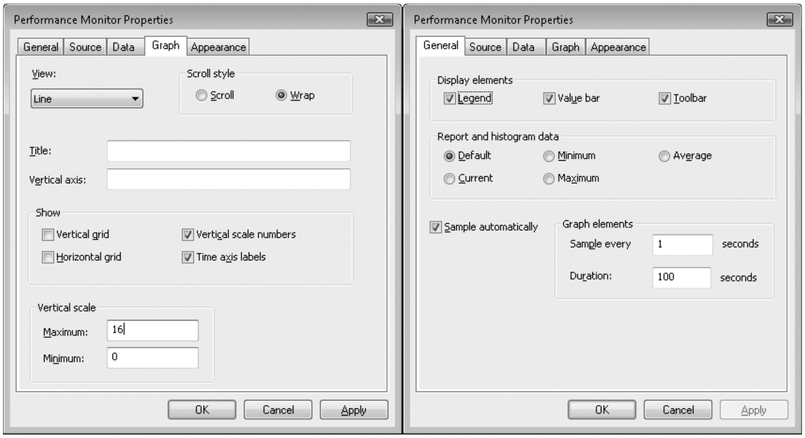
Added counters

Counter	Parent	Instance	Computer
Thread			
% Processor Time	CPUSTRES	1	

☐ Show description

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

8. Select Properties from the Action menu. Change the Vertical Scale Maximum to 16 and set the interval to Sample Every *N* Seconds in the Graph Elements area.


Performance Monitor Properties

General | Source | Data | Graph | Appearance

View: Line

Scroll style: ○ Scroll ● Wrap

Title:

Vertical axis:

Show: ☐ Vertical grid ☑ Vertical scale numbers ☐ Horizontal grid ☑ Time axis labels

Vertical scale: Maximum: 16 Minimum: 0

Performance Monitor Properties

General | Source | Data | Graph | Appearance

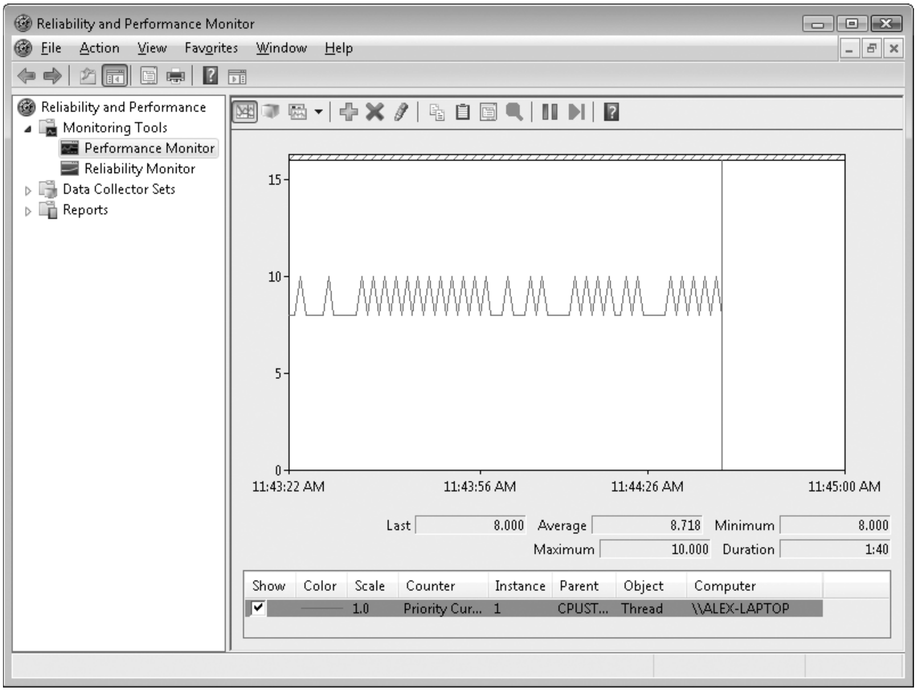
Display elements: ☑ Legend ☑ Value bar ☑ Toolbar

Report and histogram data: ● Default ○ Minimum ○ Average ○ Current ○ Maximum

☑ Sample automatically

Graph elements: 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 Reliability and 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, select the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. Be sure that the Programs option is selected. This causes *PsPrioritySeperation* to get a value of 2.
2. Run Notepad from the Start menu by selecting Programs/Accessories/Notepad.
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability And 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 then click Search. Scroll down until you see Notepad thread 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, set the interval to Sample Every *N* Seconds in the Graph Elements area, and click OK.
8. You should see the priority of thread 0 in Notepad at 8, 9, 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 9 and then to 8.
9. With Reliability and 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 Reliability and 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?

We have previously seen 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 9).

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 4 quantum units. Once the quantum is expired, 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 will return 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 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 would indicate an unusually busy system), it stops the scan at that point and picks up again on the next pass.

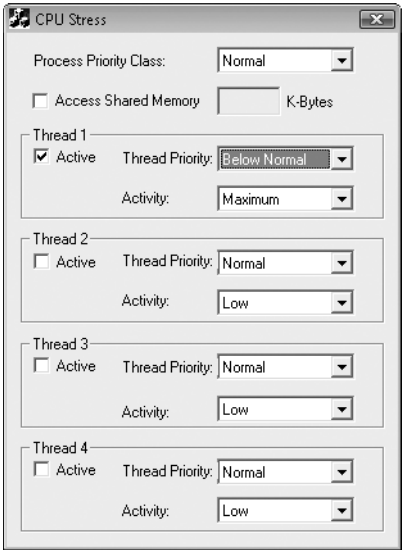
Note We mentioned earlier that scheduling decisions in Windows are not affected by the number of threads, and that they are made in *constant time*, or O(1). Because the balance set manager does need to scan ready queues manually, this operation does depend on the number of threads on the system, and 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 reenter 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, we'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

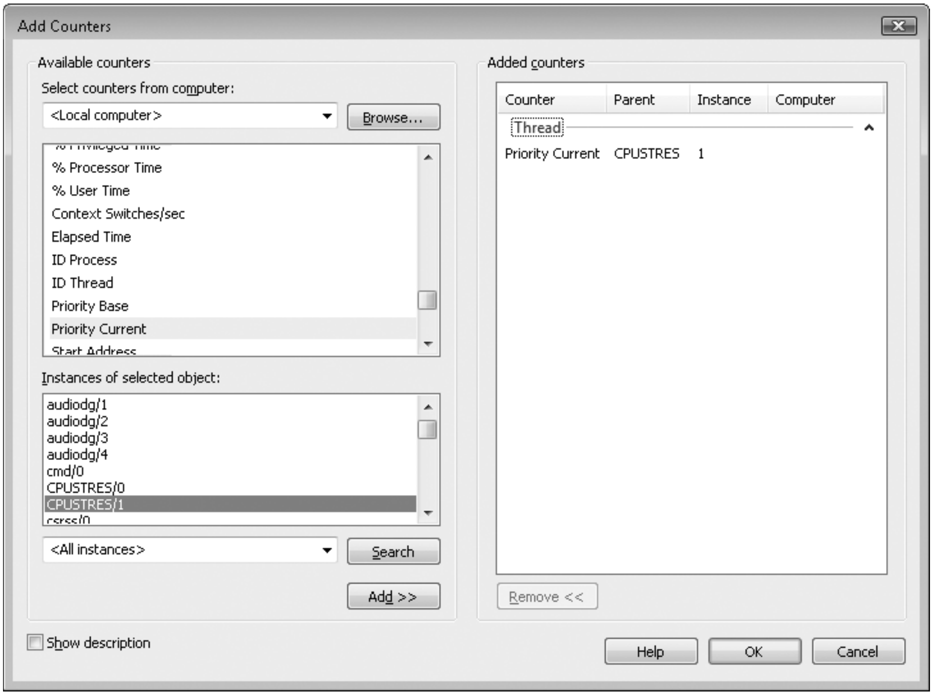
Thread 3
□ Active Thread Priority: Normal
Activity: Low

Thread 4
□ Active Thread Priority: Normal
Activity: Low

2. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability And 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+I) to bring up the Add Counters dialog box.

4. Select the Thread object, and then select the % Processor Time counter.

5. In the Instances box, select <All instances>, and then 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:



Select counters from computer: <Local computer>

Instances of selected object: CPUSTRES/1

<All instances>

□ Show description

Counter	Parent	Instance	Computer
Thread			
Priority Current	CPUSTRES	1	

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

7. Raise the priority of Performance Monitor to real time by running Task Manager, clicking 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 will 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 6 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’s priority boosts (which we will describe in the next subsection), you will 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. Raise the priority of Thread 1 from Normal to Time Critical.
5. You should hear the music playback stop as the compute-bound thread begins consuming all available CPU time.
6. 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.
7. Stop Cpustres and Windows Media Player, and start the MMCSS service again.

Priority Boosts for MultiMedia Applications and Games (MMCSS)

As we’ve just seen in the last experiment, although Windows’s CPU starvation priority boosts may 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 Vista would have only made the situation worse since it offers even more chances for preemption. To address this, Windows Vista incorporates a new 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:

- 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 5-19 shows the various scheduling categories.

TABLE 5-19 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	Threads part of a foreground application such as Windows Media Player.
Low	8-15	All other threads not part of the previous categories.
Exhausted	1-7	Threads that have exhausted their share of the CPU and will only continue running 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 will get 80 percent of the CPU time available, while other threads will receive 20 percent (based on a sample of 10 ms; in other words, 8 ms and 2 ms). MMCSS itself runs at priority 27, since it needs to preempt any Pro Audio threads in order to lower their priority to the Exhausted category.

It is important to emphasize 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 do), 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, since they are in the real-time range and well above threads that most user applications would be running 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).

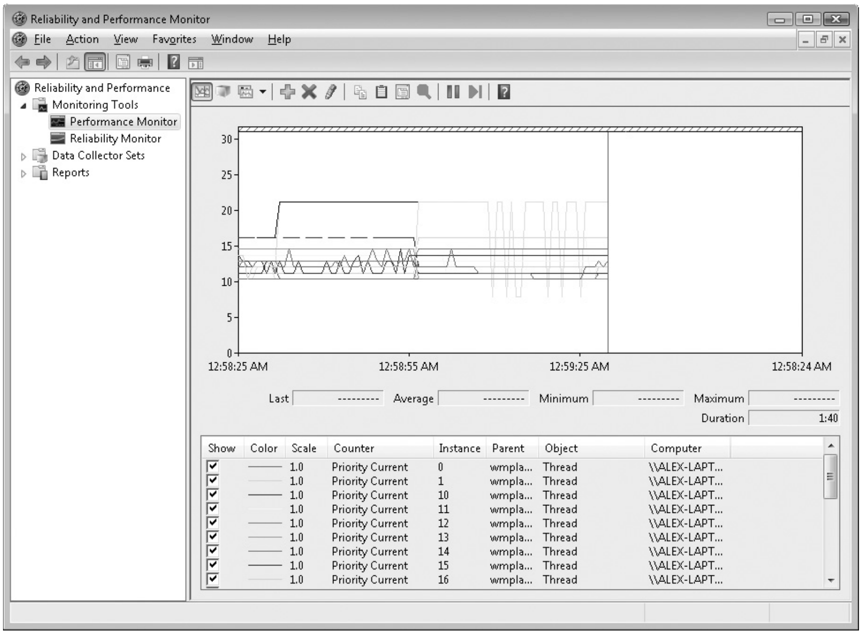
EXPERIMENT: “Listening” to MMCSS Priority Boosting

We are now going to perform the same experiment as the prior one but without disabling the MMCSS service. In addition, we’ll take a look at the Performance tool to check the priority of the Windows Media Player threads.

1. Run Windows Media Player (other playback programs may 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 only runs on one CPU (since we’ll be using only one CPUSTRES worker thread).
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability And 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 then click Search. Scroll down until you see Wmplayer, 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, set the interval to Sample Every *N* Seconds in the Graph Elements area, 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, and, 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's functionality does not stop at simple priority boosting, however. Because of the nature of network drivers on Windows and the NDIS stack, 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 could 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 will leave them out of this description.

Note The original implementation of the network throttling code had some design issues causing 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.

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, as 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, while Windows attempts to schedule the highest-priority runnable threads on all available CPUs, it only guarantees to be running the (single) highest-priority thread 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 two different types of multiprocessor systems supported by Windows (hyperthreaded, multicore, and NUMA).

Multiprocessor Considerations in the Dispatcher Database

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 "Multiprocessor Thread-Scheduling Algorithms".) Following are the two bitmasks that Windows maintains:

- The *active processor mask (KeActiveProcessors)*, which has a bit set for each usable processor on the system (This might be less than the number of actual processors if the licensing limits of the version of Windows running supports less than the number of available physical processors.)
- The *idle summary* (*KiIdleSummary*), in which each set bit represents an idle processor

Whereas on uniprocessor systems, the dispatcher database is locked by raising IRQL to both DPC/dispatch level and Synch level, on multiprocessor systems more is required, because each processor could, at the same time, raise IRQL and attempt to operate on the dispatcher database. (This is true for any systemwide structure accessed from high IRQL.) (See Chapter 3 for a general description of kernel synchronization and spinlocks.)

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 new per-PRCB queued spinlock, which is held at IRQL SYNCH_LEVEL. (See Table 5-20 for the various values of SYNCH_LEVEL.) Thus, thread selection can occur while locking only an individual processor's PRCB, in contrast to doing this on Windows XP, where the systemwide dispatcher spinlock had to be held.

TABLE 5-20 IRQL SYNCH_LEVEL on Multiprocessor Systems

CPU Type	IRQL
Systems running on x86	27
Systems running on x64	12
Systems running on IA64	12

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 before exiting the thread dispatcher, before performing a context switch, and after processing a DPC. Threads on the deferred ready list are either dispatched immediately or are moved to the per-processor ready queue for their priority level.

Note that the systemwide dispatcher spinlock still exists and is used, but it is held only for the time needed to modify systemwide state that might affect which thread runs next. For example, changes to synchronization objects (mutexes, events, and semaphores) and their wait queues require holding the dispatcher lock to prevent more than one processor from changing the state of such objects (and the consequential action of possibly readying threads for execution). Other examples include changing the priority of a thread, timer expiration, and swapping of thread kernel stacks.

Thread context switching is also synchronized by using a finer-grained per-thread spinlock, whereas in Windows XP context switching was synchronized by holding a systemwide context swap spinlock.

Hyperthreaded and Multicore Systems

As described in the "Symmetric Multiprocessing" section in Chapter 2, Windows supports hyperthreaded and multicore multiprocessor systems in two primary ways:

1. Logical processors as well as per-package cores do not count against physical processor licensing limits. For example, Windows Vista Home Basic, which has a licensed processor limit of 1, will use all four cores on a single processor system.
2. When choosing a processor for a thread, if there is a physical processor with all logical processors idle, a logical processor from that physical processor will be selected, as opposed to choosing an idle logical processor on a physical processor that has another logical processor running a thread.

EXPERIMENT: Viewing Hyperthreading Information

You can examine the information Windows maintains for hyperthreaded processors using the *!smt* command in the kernel debugger. The following output is from a dual-processor hyperthreaded Xeon system (four logical processors):

```
lkd> !smt
SMT Summary:
------------
   KeActiveProcessors: ****---------------------------- (0000000f)
        KiIdleSummary: -***---------------------------- (0000000e)
No PRCB     Set Master SMT Set                                     #LP IAID
 0 ffdff120 Master     *-*----------------------------- (00000005)   2  00
 1 f771f120 Master     -*-*---------------------------- (0000000a)   2  06
 2 f7727120 ffdff120   *-*----------------------------- (00000005)   2  01
 3 f772f120 f771f120   -*-*---------------------------- (0000000a)   2  07

   Number of licensed physical processors: 2
```

Logical processors 0 and 1 are on separate physical processors (as indicated by the term “Master”).

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. While 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
nt!_KNODE
   +0x000 PagedPoolSListHead : _SLIST_HEADER
   +0x008 NonPagedPoolSListHead : [3] _SLIST_HEADER
   +0x020 PfnDereferenceSListHead : _SLIST_HEADER
   +0x028 ProcessorMask    : Uint4B
   +0x02c Color            : UChar
   +0x02d Seed             : UChar
   +0x02e NodeNumber       : UChar
   +0x02f Flags            : _flags
   +0x030 MmShiftedColor   : Uint4B
   +0x034 FreeCount        : [2] Uint4B
   +0x03c PfnDeferredList  : Ptr32 _SINGLE_LIST_ENTRY
   +0x040 CachedKernelStacks : _CACHED_KSTACK_LIST
```

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 32-processor NUMA system by NEC with 4 processors per node:

```
21: kd> !numa
NUMA Summary:
------------
Number of NUMA nodes : 8
Number of Processors : 32
MmAvailablePages     : 0x00F70D2C
KeActiveProcessors   : ********************************--------------------------------
                       (00000000ffffffff)

NODE 0 (E00000008428AE00):
    ProcessorMask    : ****------------------------------------------------------------
    Color            : 0x00000000
    MmShiftedColor   : 0x00000000
    Seed             : 0x00000000
    Zeroed Page Count: 0x00000000001CF330
    Free Page Count  : 0x0000000000000000

NODE 1 (E00001597A9A2200):
    ProcessorMask    : ----****--------------------------------------------------------
    Color            : 0x00000001
    MmShiftedColor   : 0x00000040
    Seed             : 0x00000006
    Zeroed Page Count: 0x00000000001F77A0
    Free Page Count  : 0x0000000000000004
```

The following partial output is from a 64-processor NUMA system from Hewlett-Packard with 4 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    : ****------------------------------------------------------------
    Color            : 0x00000000
    MmShiftedColor   : 0x00000000
    Seed             : 0x00000001
    Zeroed Page Count: 0x00000000003F4430
    Free Page Count  : 0x0000000000000000
```

```
NODE 1 (E0000145FF992200):
    ProcessorMask    : ----****--------------------------------------------------------
    Color            : 0x00000001
    MmShiftedColor   : 0x00000040
    Seed             : 0x00000007
    Zeroed Page Count: 0x00000000003ED59A
    Free Page Count  : 0x0000000000000000
```

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. This information can be obtained using the functions listed in Table 5-21. Functions that can alter thread affinity are listed in Table 5-13.

TABLE 5-21 NUMA-Related Functions

Function	Description
GetNumaHighestNodeNumber	Retrieves the node that currently has the highest number.
GetNumaNodeProcessorMask	Retrieves the processor mask for the specified node.
GetNumaProximityNode	Returns the NUMA node number for the given proximity ID.
GetNumaProcessorNode	Retrieves the node number for the specified processor.

How the scheduling algorithms take into account NUMA systems will be covered in the upcoming section “Multiprocessor Thread-Scheduling Algorithms” (and the optimizations in the memory manager to take advantage of node-local memory are covered in Chapter 9).

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 active processors on the system—in other words, the system is free to schedule all threads on any available processor.

However, to optimize throughput and/or partition workloads to a specific set of processors, 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.)

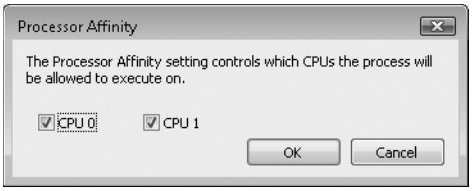
- By making a process a member of a job that has a jobwide affinity mask set using the *SetInformationJobObject* function (Jobs are described in the upcoming “Job Objects” section.)
- By specifying an affinity mask in the image header when compiling the application (For more information on the detailed format of Windows images, search for “Portable Executable and Common Object File Format Specification” on *www.microsoft.com*.)

You can also set the “uniprocessor” flag for an image (at compile time). If this flag is set, the system chooses a single processor at process creation time and assigns that as the process affinity mask, starting with the first processor and then going round-robin across all the processors. For example, on a dual-processor system, the first time you run an image marked as uniprocessor, 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. (This has actually saved the authors of this book on two different occasions.)

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 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’s 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 wait.

Therefore, changing the affinity mask for a process or a thread can result in threads getting less CPU time than they normally would, as 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.

Ideal and Last Processor

Each thread has two CPU numbers stored in the kernel thread block:

- *Ideal processor,* or the preferred processor that this thread should run on
- *Last processor,* or the processor on which the thread last ran

The ideal processor for a thread is chosen when a thread is created using a seed in the process block. The seed is incremented each time a thread is created so that the ideal processor for each new thread in the process will rotate 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 evenly 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.

On hyperthreaded systems, the next ideal processor is the first logical processor on the next physical processor. For example, on a dual-processor hyperthreaded 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.

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's 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.

Dynamic Processor Addition and Replacement

As we'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 of the times 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 will notify 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, a variety of 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 Descriptor Table (IDT), the processor control region (PCR), the processor 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 lookaside lists for the processor that was added. For example, the I/O manager, the executive lookaside list code, the cache manager, and the object manager all use per-processor lookaside 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, the processor add 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. Unfortunately, until now, CPU-hungry applications have still been left out of this process, but Windows Server 2008 and Windows Vista Service Pack 1 have improved the process to allow applications to be 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 (since 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.

Multiprocessor Thread-Scheduling Algorithms

Now that we've described the types of multiprocessor systems supported by Windows as well as the thread affinity and ideal processor settings, we're ready to examine how this information is used to determine which threads run where. There are two basic decisions to describe:

- Choosing a processor for a thread that wants to run
- Choosing a thread on a processor that needs something to do

Choosing a Processor for a Thread When There Are Idle Processors

When a thread becomes ready to run, Windows first tries to schedule the thread to run on an idle processor. If there is a choice of idle processors, preference is given first to the thread's ideal processor, then to the thread's previous processor, and then to the currently executing processor (that is, the CPU on which the scheduling code is running).

To select the best idle processor, Windows starts with the set of idle processors that the thread's affinity mask permits it to run on. If the system is NUMA and there are idle CPUs in the node containing the thread's ideal processor, the list of idle processors is reduced to that set. If this eliminates all idle processors, the reduction is not done. Next, if the system is running hyperthreaded processors and there is a physical processor with all logical processors idle, the list of idle processors is reduced to that set. If that results in an empty set of processors, the reduction is not done.

If the current processor (the processor trying to determine what to do with the thread that wants to run) is in the remaining idle processor set, the thread is scheduled on it. If the current processor is not in the remaining set of idle processors, it is a hyperthreaded system, and there is an idle logical processor on the physical processor containing the ideal processor for the thread, the idle processors are reduced to that set. If not, the system checks whether there are any idle logical processors on the physical processor containing the thread's previous processor. If that set is nonzero, the idle processors are reduced to that list. Finally, the lowest numbered CPU in the remaining set is selected as the processor to run the thread on.

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. When the idle loop on that processor runs, it will see that a thread has been selected to run and will dispatch that thread.

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

If there are no idle processors when a thread wants to run, 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 an interprocessor interrupt to the target processor to preempt the currently running thread in favor of this new thread.

> **Note** Windows doesn't look at the priority of the current and next threads on all the CPUs—just on the one CPU selected as just described. If no thread can be preempted on that one CPU, the new thread is put in the ready queue for its priority level, where it awaits its turn to get scheduled. Therefore, Windows does not guarantee to be running all the highest-priority threads, but it will always run the highest-priority thread.

If the ready thread cannot be run right away, it is moved into the ready state where it awaits its turn to run. Note that threads are always put on their ideal processor's per-processor ready queues.

Selecting a Thread to Run on a Specific CPU

Because each processor has its own list of threads waiting to run on that processor, when a thread finishes running, the processor can simply check its per-processor ready queue for the next thread to run. If the per-processor ready queues are empty, the idle thread for that processor is scheduled. The idle thread then begins scanning other processor's ready queues for threads it can run. Note that on NUMA systems, the idle thread first looks at processors on its node before looking at other nodes' processors.

CPU Rate Limits

As part of the new hard quota management system added in Windows Vista (which builds on previous quota support present since the first version of Windows NT, but adds hard limits instead of soft hints), support for limiting CPU usage was added to the system in three different ways: per-session, per-user, or per-system. Unfortunately, information on enabling these new limits has not yet been documented, and no tool that is part of the operating system 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 9, which deals with the memory manager, and focus our attention on the CPU rate limit.

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 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 value inside that key.
- By calling *NtSetInformationProcess* and giving it the process handle of the process to limit and the CPU rate limiting information.

In all three cases, the CPU rate limit data is not a straightforward value; it is based on a compressed bitfield, documented in the WDK as part of the RATE_QUOTA_LIMIT structure. The bottom four bits define the *rate phase*, which can be expressed either as one, two, or three seconds—this value defines how often the rate limiting should be applied and is called the PS_RATE_PHASE. The rest of the bits are used for the actual rate, as a value representing a percentage of maximum CPU usage. Because any number from 0 to 100 can be represented with only 7 bits, the rest of the bits are unused. Therefore, a rate limit of 40 percent every 2 seconds would be defined by the value 0x282, or 101000 0010 in binary.

The process manager, which is responsible for enforcing the CPU rate limit, uses a variety of system mechanisms to do its job. First of all, rate limiting is able to reliably work 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 a kernel gate object (making the thread uniquely bound to this object and putting it in a wait state, which does not consume CPU cycles). 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 gate is signaled by an internal worker thread inside the process manager responsible for replenishment of the CPU usage, which is queued by a DPC responsible for replenishing systemwide CPU usage requests.

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 descendents 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. Table 5-22 lists the Windows functions to create and manipulate job objects.

TABLE 5-22 Windows API Functions for Jobs

Function	Description
CreateJobObject	Creates a job object (with an optional name)
OpenJobObject	Opens an existing job object by name
AssignProcessToJobObject	Adds a process to a job
TerminateJobObject	Terminates all processes in a job
SetInformationJobObject	Sets limits
QueryInformationJobObject	Retrieves information about the job, such as CPU time, page fault count, number of processes, list of process IDs, quotas or limits, and security limits

The following are some of the CPU-related and memory-related limits you can specify for a job:

- **Maximum number of active processes** Limits the number of concurrently existing processes in the job.
- **Jobwide user-mode CPU time limit** Limits the maximum amount of user-mode CPU time that the processes in the job can consume (including processes that have run and exited). Once this limit is reached, by default all the processes in the job will be terminated with an error code and no new processes can be created in the job (unless the limit is reset). The job object is signaled, so any threads waiting for the job will be released. You can change this default behavior with a call to *EndOfJobTimeAction*.
- **Per-process user-mode CPU time limit** Allows each process in the job to accumulate only a fixed maximum amount of user-mode CPU time. When the maximum is reached, the process terminates (with no chance to clean up).
- **Job scheduling class** Sets the length of the time slice (or quantum) for threads in processes in the job. This setting applies only to systems running with long, fixed quantums (the default for Windows Server systems). The value of the job-scheduling class determines the quantum as shown here:

Scheduling Class	Quantum Units
0	6
1	12
2	18
3	24
4	30
5	36
6	42
7	48
8	54
9	Infinite if real-time; 60 otherwise

- **Job processor affinity** Sets the processor affinity mask for each process in the job. (Individual threads can alter their affinity to any subset of the job affinity, but processes can't alter their process affinity setting.)
- **Job process priority class** Sets the priority class for each process in the job. Threads can't increase their priority relative to the class (as they normally can). Attempts to increase thread priority are ignored. (No error is returned on calls to *SetThreadPriority*, but the increase doesn't occur.)
- **Default working set minimum and maximum** Defines the specified working set minimum and maximum for each process in the job. (This setting isn't jobwide—each process has its own working set with the same minimum and maximum values.)
- **Process and job committed virtual memory limit** Defines the maximum amount of virtual address space that can be committed by either a single process or the entire job.

Jobs can also be set to queue an entry to an I/O completion port object, which other threads might be waiting for, with the Windows *GetQueuedCompletionStatus* function.

You can also place security limits on processes in a job. You can set a job so that each process runs under the same jobwide access token. You can then create a job to restrict processes from impersonating or creating processes that have access tokens that contain the local administrator's group. In addition, you can apply security filters so that when threads in processes contained in a job impersonate client threads, certain privileges and security IDs (SIDs) can be eliminated from the impersonation token.

Finally, you can also place user-interface limits on processes in a job. Such limits include being able to restrict processes from opening handles to windows owned by threads outside the job, reading and/or writing to the clipboard, and changing the many user-interface system parameters via the Windows *SystemParametersInfo* function.

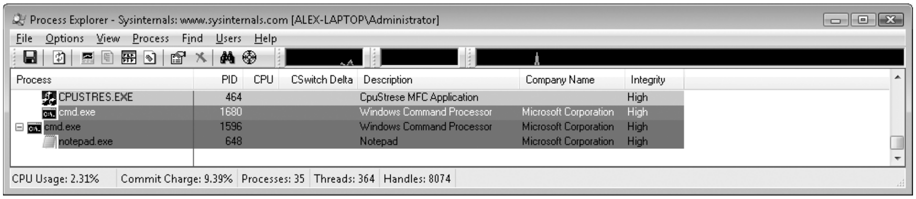
EXPERIMENT: Viewing the Job Object

You can view named job objects with the Performance tool. (See the Job Object and Job Object Details performance objects.) You can view unnamed jobs with the kernel debugger *!job* or *dt nt!_ejob* commands.

To see whether a process is associated with a job, you can use the kernel debugger *!process* command or Process Explorer. Follow these steps to create and view an unnamed job object:

1. From the command prompt, use the *runas* command to create a process running the command prompt (Cmd.exe). For example, type **runas /user:<domain>\< username> cmd**. You'll be prompted for your password. Enter your password, and a Command Prompt window will appear. The Windows service that executes runas commands creates an unnamed job to contain all processes (so that it can terminate these processes at logoff time).

2. From the command prompt, run Notepad.exe.
3. Then run Process Explorer and notice that the Cmd.exe and Notepad.exe processes are highlighted as part of a job. (You can configure the colors used to highlight processes that are members of a job by clicking Options, Configure Highlighting.) Here is a screen shot showing these two processes:

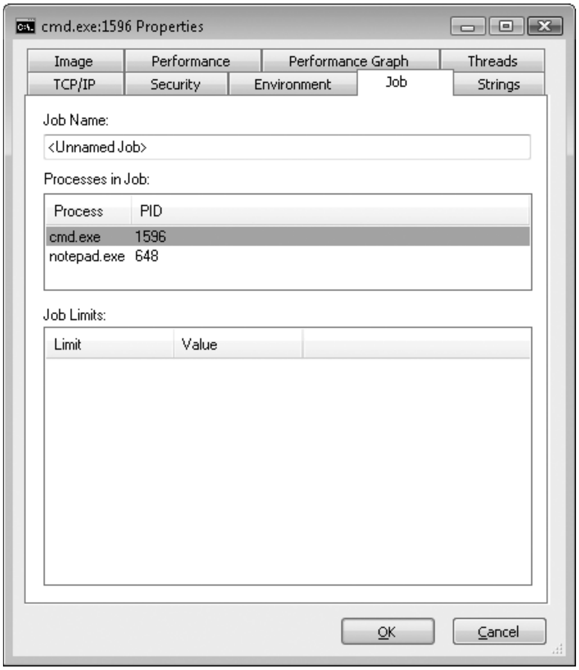


Process Explorer - Sysinternals: www.sysinternals.com [ALEX-LAPTOP\Administrator]
File
Options
View
Process
Find
Users
Help

Process	PID	CPU	CSwitch Delta	Description	Company Name	Integrity
CPUSTRES.EXE	464			CpuStrese MFC Application		High
cmd.exe	1680			Windows Command Processor	Microsoft Corporation	High
cmd.exe	1596			Windows Command Processor	Microsoft Corporation	High
notepad.exe	648			Notepad	Microsoft Corporation	High

CPU Usage: 2.31% Commit Charge: 9.39% Processes: 35 Threads: 364 Handles: 8074

4. Double-click either the Cmd.exe or Notepad.exe process to bring up the process properties. You will see a Job tab in the process properties dialog box.
5. Click the Job tab to view the details about the job. In this case, there are no quotas associated with the job, but there are two member processes:



Job Name:

<Unnamed Job>

Processes in Job:

Process	PID
cmd.exe	1596
notepad.exe	648

Job Limits:

Limit	Value

6. Now run the kernel debugger on the live system, display the process list with *!process*, and find the recently created process running Cmd.exe. Then display the process block by using *!process <process ID>*, find the address of the job object, and finally display the job object with the *!job* command. Here's some partial debugger output of these commands on a live system:

   ```
   lkd> !process 0 0
   **** NT ACTIVE PROCESS DUMP ****
       .
       .
   PROCESS 8567b758  SessionId: 0  Cid: 0fc4    Peb: 7ffdf000  ParentCid: 00b0
       DirBase: 1b3fb000  ObjectTable: e18dd7d0  HandleCount:  19.
       Image: Cmd.exe

   PROCESS 856561a0  SessionId: 0  Cid: 0d70    Peb: 7ffdf000  ParentCid: 0fc4
       DirBase: 2e341000  ObjectTable: e19437c8  HandleCount:  16.
       Image: Notepad.exe

   lkd> !process 0fc4
   Searching for Process with Cid == fc4
   PROCESS 8567b758  SessionId: 0  Cid: 0fc4    Peb: 7ffdf000  ParentCid: 00b0
       DirBase: 1b3fb000  ObjectTable: e18dd7d0  HandleCount:  19.
       Image: Cmd.exe
       BasePriority                      8
       .
       .
       Job                               85557988

   lkd> !job 85557988
   Job at 85557988
     TotalPageFaultCount      0
     TotalProcesses           2
     ActiveProcesses          2
     TotalTerminatedProcesses 0
     LimitFlags               0
     MinimumWorkingSetSize    0
     MaximumWorkingSetSize    0
     ActiveProcessLimit       0
     PriorityClass            0
     UIRestrictionsClass      0
     SecurityLimitFlags       0
     Token                    00000000
   ```

7. Finally, use the *dt* command to display the job object and notice the additional fields shown about the job:

   ```
   lkd> dt nt!_ejob 85557988
   nt!_EJOB
      +0x000 Event            : _KEVENT
      +0x010 JobLinks         : _LIST_ENTRY [ 0x81d09478 - 0x87f55030 ]
      +0x018 ProcessListHead  : _LIST_ENTRY [ 0x87a08dd4 - 0x8679284c ]
      +0x020 JobLock          : _ERESOURCE
      +0x058 TotalUserTime    : _LARGE_INTEGER 0x0
   ```

```
   +0x060 TotalKernelTime  : _LARGE_INTEGER 0x0
   +0x068 ThisPeriodTotalUserTime : _LARGE_INTEGER 0x0
   +0x070 ThisPeriodTotalKernelTime : _LARGE_INTEGER 0x0
   +0x078 TotalPageFaultCount : 0
   +0x07c TotalProcesses   : 2
   +0x080 ActiveProcesses  : 2
   +0x084 TotalTerminatedProcesses : 0
   +0x088 PerProcessUserTimeLimit : _LARGE_INTEGER 0x0
   +0x090 PerJobUserTimeLimit : _LARGE_INTEGER 0x0
   +0x098 LimitFlags       : 0
   +0x09c MinimumWorkingSetSize : 0
   +0x0a0 MaximumWorkingSetSize : 0
   +0x0a4 ActiveProcessLimit : 0
   +0x0a8 Affinity         : 0
   +0x0ac PriorityClass    : 0 ''
   +0x0b0 AccessState      : (null)
   +0x0b4 UIRestrictionsClass : 0
   +0x0b8 EndOfJobTimeAction : 0
   +0x0bc CompletionPort   : 0x87e3d2e8
   +0x0c0 CompletionKey    : 0x07a89508
   +0x0c4 SessionId        : 1
   +0x0c8 SchedulingClass  : 5
   +0x0d0 ReadOperationCount : 0
   +0x0d8 WriteOperationCount : 0
   +0x0e0 OtherOperationCount : 0
   +0x0e8 ReadTransferCount : 0
   +0x0f0 WriteTransferCount : 0
   +0x0f8 OtherTransferCount : 0
   +0x100 ProcessMemoryLimit : 0
   +0x104 JobMemoryLimit   : 0
   +0x108 PeakProcessMemoryUsed : 0x19e
   +0x10c PeakJobMemoryUsed : 0x2ed
   +0x110 CurrentJobMemoryUsed : 0x2ed
   +0x114 MemoryLimitsLock : _EX_PUSH_LOCK
   +0x118 JobSetLinks      : _LIST_ENTRY [ 0x8575cff0 - 0x8575cff0 ]
   +0x120 MemberLevel      : 0
   +0x124 JobFlags         : 0
```

Conclusion

In this chapter, we've examined the structure of processes and threads and jobs, seen how they are created, and looked at how Windows decides which threads should run and for how long.

In the next chapter we'll look at a part of the system that's received more attention in the last few years than ever before, Windows security.