Chapter 6

Processes, Threads, and Jobs

In this chapter, we'll explain the data structures and algorithms that deal with processes, threads, and jobs in Microsoft Windows. 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 space, with the exception of the process environment block (PEB), which exists in the process address space (because it contains information that is modified by user-mode code).

In addition to the EPROCESS block, the Windows subsystem process (Csrss) maintains a parallel structure for each Windows process that executes a Windows program. Also, the kernel-mode part of the Windows subsystem (Win32k.sys) has 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 6-1 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.

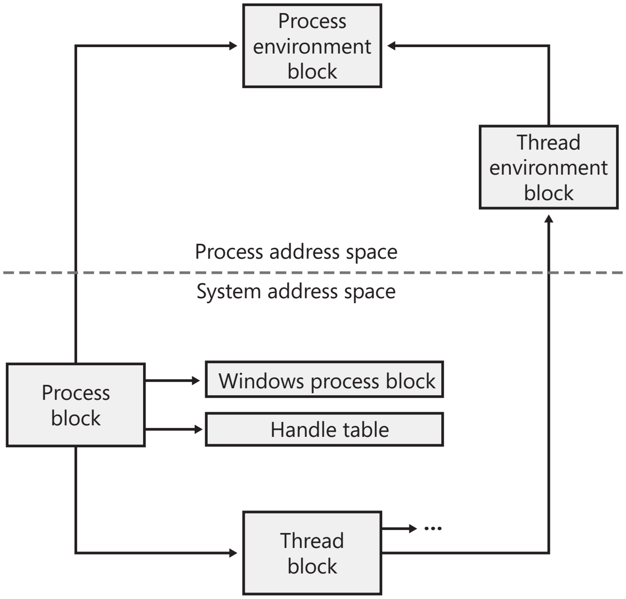


Figure 6-1 Data structures associated with processes and threads

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 6-2 shows the key fields in an EPROCESS block.

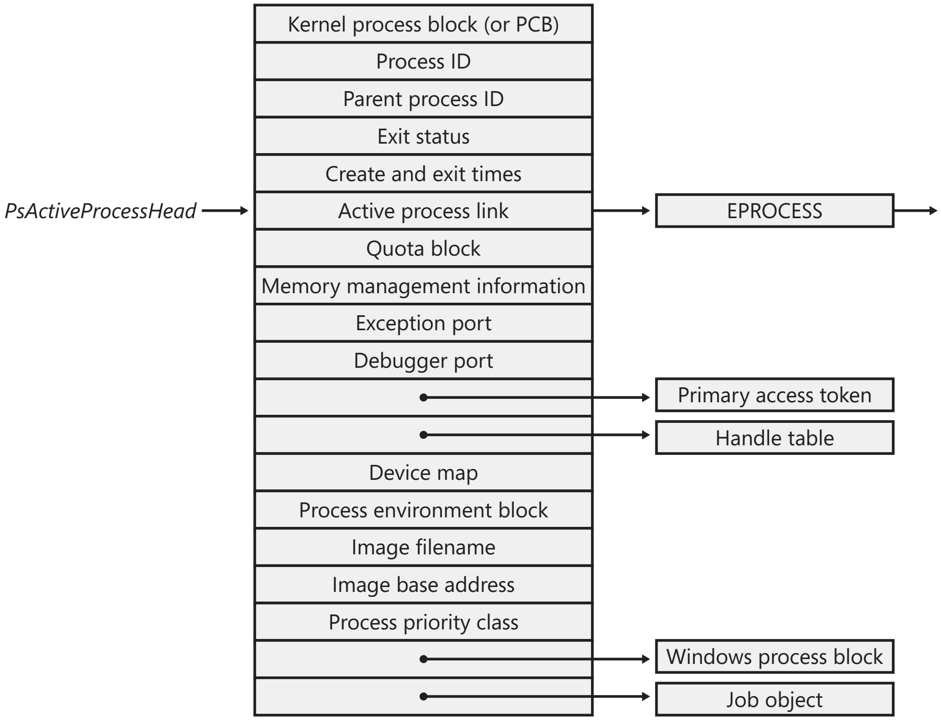


Figure 6-2 Structure of an executive process block

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) looks like this:

```
lkd> dt _eprocess
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
```

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 : [2] 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 StackCount       : Uint2B
   +0x062 BasePriority     : Char
   +0x063 ThreadQuantum    : Char
   +0x064 AutoAlignment    : UChar
   +0x065 State            : UChar
   +0x066 ThreadSeed       : UChar
   +0x067 DisableBoost     : UChar
   +0x068 PowerState       : UChar
   +0x069 DisableQuantum   : UChar
   +0x06a IdealNode        : UChar
   +0x06b Spare            : UChar
```

An alternate 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 6-1 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 an integral part 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 6-1 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, quantum, affinity mask, and total kernel and user time for the threads in the process.	Thread Scheduling (Chapter 6)
Process identification	Unique process ID, creating process ID, name of image being run, window station process is running on.	
Quota block	Limits on 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 point to the single systemwide default quota block; all the processes in the interactive session share a single quota block that Winlogon sets up.)	
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 7)
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 7)
Virtual memory information	Current and peak virtual size, page file usage, hardware page table entry for process page directory.	Chapter 7
Exception 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)

Table 6-1 Contents of the EPROCESS Block

Element	Purpose	Additional Reference
Debugging LPC port	Interprocess communication channel to which the process manager sends a message when one of the process's threads causes a debug event.	Local Procedure Calls (LPCs) (Chapter 3)
Access token (ACCESS_TOKEN)	Executive object describing the security profile of this process.	Chapter 8
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 6
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 6-3. It contains the basic information that the Windows kernel needs to schedule threads. (Page directories are covered in Chapter 7, 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 modify 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 6-4 and is explained in more detail later in this chapter.

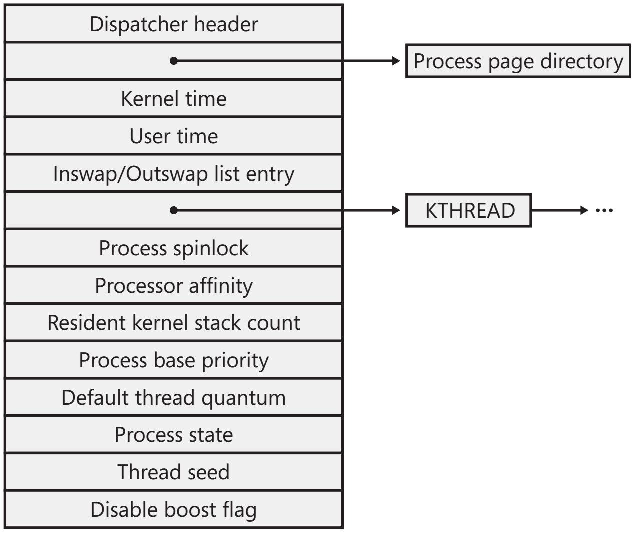


Figure 6-3 Structure of the executive process block

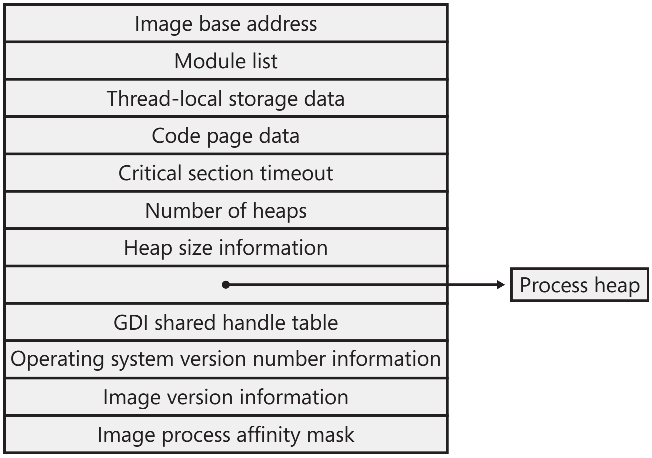


Figure 6-4 Fields of the process environment block

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: 0  Cid: 08d0    Peb: 7ffdf000  ParentCid: 0360
    DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  66.
    Image: windbg.exe
```

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

```
lkd> !peb 7ffdf000
PEB at 7ffdf000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         01000000
    Ldr                       00181e90
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00181f28 . 00183188
    Ldr.InLoadOrderModuleList:           00181ec0 . 00183178
    Ldr.InMemoryOrderModuleList:         00181ec8 . 00183180
            Base TimeStamp                     Module
         1000000 40478dbd Mar 04 15:12:45 2004 C:\Program Files\Debugging Tools for
             Windows\windbg.exe
        77f50000 3eb1b41a May 01 19:56:10 2003 C:\WINDOWS\System32\ntdll.dll
        77e60000 3d6dfa28 Aug 29 06:40:40 2002 C:\WINDOWS\system32\kernel32.dll
         2000000 40476db2 Mar 04 12:56:02 2004 C:\Program Files\Debugging Tools for
             Windows\dbgeng.dll
        .
    SubSystemData:     00000000
    ProcessHeap:       00080000
    ProcessParameters: 00020000
    WindowTitle:  'C:\Documents and Settings\All Users\Start Menu\Programs\Debugging
        Tools for Windows\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:\Program Files\Windows Resource Kits\Tools\;C:\WINDOWS\
system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\Support Tools\;c:\sysint
;C:\Program Files\ATI Technologies\ATI Control Panel;C:\Program Files\Resource Kit\;C:
\PROGRA~1\CA\Common\SCANEN~1;C:\PROGRA~1\CA\eTrust\ANTIVI~1;C:\Program Files\Common
    Files\Roxio Shared\DLLShared;C:\SFU\common\'
    Environment:  00010000
        =::=::\
        ALLUSERSPROFILE=C:\Documents and Settings\All Users
        APPDATA=C:\Documents and Settings\dsolomon\Application Data
    .
    .
    .
```

Kernel Variables

A few key kernel global variables that relate to processes are listed in Table 6-2. These variables are referred to later in the chapter, when the steps in creating a process are described.

Table 6-2 Process-Related Kernel Variables

Variable	Type	Description
PsActiveProcessHead	Queue header	List head of process blocks
PsIdleProcess	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 pointers	Array of pointers to routines to be called on process creation and deletion (maximum of eight)
PspCreateProcessNotifyRoutineCount	DWORD	Count of registered process notification routines
PspLoadImageNotifyRoutine	Array of pointers	Array of pointers to routines to be called on image load
PspLoadImageNotifyRoutineCount	DWORD	Count of registered image-load notification routines
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 6-3 lists the performance counters relevant to processes (except for memory management and I/O-related counters, which are described in Chapters 7 and 9, respectively).

Table 6-3 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.
Process: % User Time	Describes the percentage of time that the threads in the process have run in user mode during a specified interval.

Table 6-3 Process-Related Performance Counters

Object: Counter	Function
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 6-4. For further information, consult the Windows API documentation in the MSDN Library.

Table 6-4 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
GetProcessTimes	Obtains a process's timing information, describing how much time the process has spent in user and kernel mode
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
GetCurrentProcess	Returns a pseudo handle for the current process
GetCurrentProcessId	Returns the ID of the current process

Table 6-4 Process-Related Functions

Function	Description
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
GetEnvironmentVariable	Returns a specific environment variable
Get/SetProcessShutdownParameters	Defines the shutdown priority and number of retries for the current process
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 8575f030  SessionId: 0  Cid: 08d0    Peb: 7ffdf000  ParentCid: 0360
    DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  65.
    Image: windbg.exe
    VadRoot 857f05e0 Vads 71 Clone 0 Private 1152. Modified 98. Locked 1.
    DeviceMap e1e96c88
    Token                             e1f5b8a8
    ElapsedTime                       1:23:06.0219
    UserTime                          0:00:11.0897
    KernelTime                        0:00:07.0450
    QuotaPoolUsage[PagedPool]         38068
    QuotaPoolUsage[NonPagedPool]      2840
    Working Set Sizes (now,min,max)  (2552, 50, 345) (10208KB, 200KB, 1380KB)
    PeakWorkingSetSize                2715
    VirtualSize                       41 Mb
    PeakVirtualSize                   41 Mb
    PageFaultCount                    3658
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      1566
```

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 8.

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 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, the Windows executive, and the Windows subsystem process (Csrss). Because of the multiple environment subsystem architecture of Windows, creating a Windows executive process object (which other subsystems can use) is separated from the work involved in creating a Windows 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 a Windows 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.

> **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 7.

1. Open the image file (.exe) to be executed inside the process.
2. Create the Windows executive process object.
3. Create the initial thread (stack, context, and Windows executive thread object).
4. Notify the Windows subsystem of the new process so that it can set up for the new process and thread.
5. Start execution of the initial thread (unless the CREATE_ SUSPENDED flag was specified).
6. 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 6-5 shows an overview of the stages Windows follows to create a process.

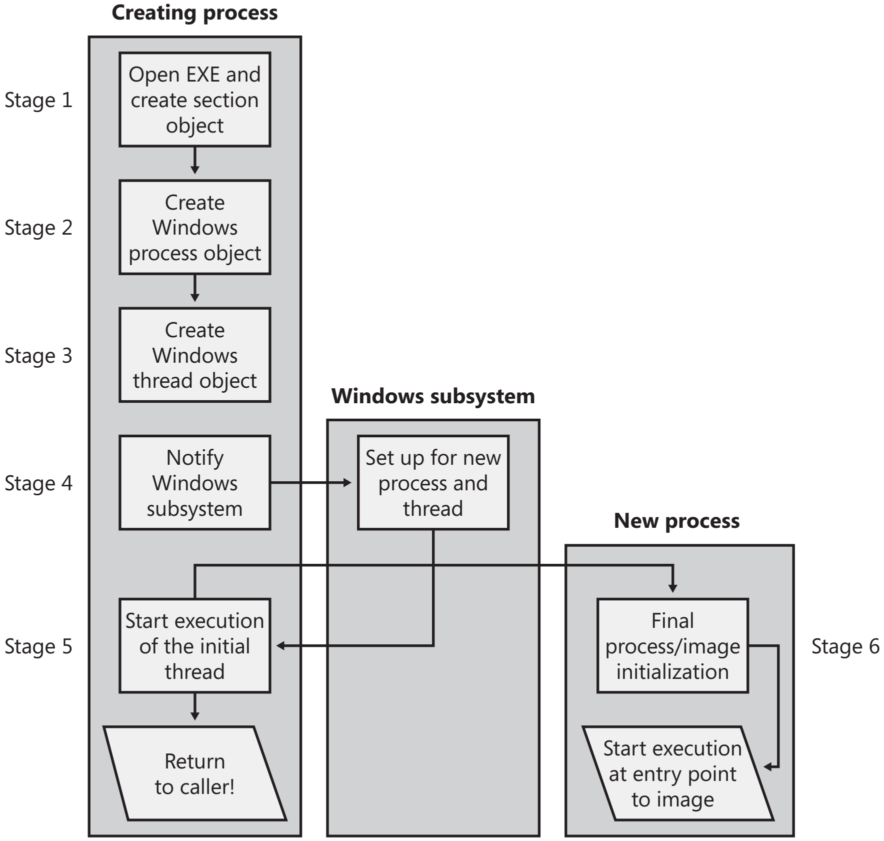

Creating process
Stage 1
Open EXE and create section object
Stage 2
Create Windows process object
Stage 3
Create Windows thread object
Stage 4
Notify Windows subsystem
Windows subsystem
Set up for new process and thread
New process
Stage 5
Start execution of the initial thread
Final process/image initialization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

Figure 6-5 The main stages of process creation

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.

Stage 1: Opening the Image to Be Executed

As illustrated in Figure 6-6, the first stage in *CreateProcess* 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 no image name is specified, the first token of the command line (defined to be the first part of the command-line string ending with a space or tab that is a valid file specification) is used as the image filename.

On Windows XP and Windows Server 2003, *CreateProcess* checks whether software restriction policies on the machine prevent the image from being run. (See Chapter 8 for a complete description of software restriction policies.)

If the executable file specified is a Windows .exe, it is used directly. If it's 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 on 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 6-5), *CreateProcess* fails.

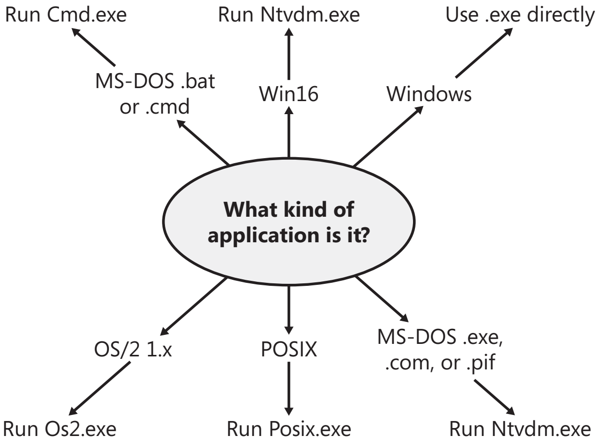


Figure 6-6 Choosing a Windows image to activate

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

If the image is a/an	And this will happen	This image will run
POSIX executable file	Posix.exe	*CreateProcess* restarts Stage 1.
MS-DOS application with an .exe, a .com, or a .pif extension	Ntvdm.exe	*CreateProcess* restarts Stage 1.
Win16 application	Ntvdm.exe	*CreateProcess* restarts Stage 1.
Command procedure (application with a .bat or a .cmd extension)	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 session-wide 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.

At this point, *CreateProcess* has successfully opened a valid Windows executable file and created a section object for it. The object isn't mapped into memory yet, but it is open. 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 *CreateProcess* has found a valid Windows executable image, it looks in the registry under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options to see whether a subkey with the filename and extension of the executable image (but without the directory and path information—for example, Image.exe) exists there. If it does, *CreateProcess* looks for a value named Debugger for that key. If this 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 *CreateProcess* behavior and debug the startup code of Windows service processes before they start rather than attach the debugger after starting the service, which doesn't allow you to debug the startup code.

Stage 2: Creating the Windows Executive Process Object

At this point, *CreateProcess* 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 *NtCreateProcess*. 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)
- 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)
- Setting up the PEB
- Completing the setup of the executive process object

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 2A: Setting Up the EPROCESS Block

This substage involves nine steps:

1. Allocate and initialize the Windows EPROCESS block.
2. Inherit the process affinity mask from the parent process.
3. The process minimum and maximum working set size are set to the values of *PsMinimumWorkingSet* and *PsMaximumWorkingSet*, respectively.
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.
5. Inherit the Windows device name space (including the definition of drive letters, COM ports, and so on).
6. Store the parent process's process ID in the *InheritedFromUniqueProcessId* field in the new process object.
7. 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.
8. 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.)
9. Set the new process's exit status to STATUS_PENDING.

Stage 2B: 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.

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

Stage 2C: Creating the Kernel Process Block

The next stage of *CreateProcess* is the initialization of the KPROCESS block, which contains a pointer to a list of kernel threads. (The kernel has no knowledge of handles, so it bypasses the object table.) The kernel process block also points 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 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, and the initial value of the process default quantum (which is described in more detail in the "Thread Scheduling" section later in the chapter), which is taken from the value of *PspForegroundQuantum[0]*, the first entry in the systemwide quantum array.

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 2D: 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 7.

- 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.
- The systemwide national language support (NLS) tables are mapped into the process's address space.

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 2E: Setting Up the PEB

CreateProcess allocates a page for the PEB and initializes a number of fields, which are described in Table 6-6.

Table 6-6 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
MaximumNumberOfHeaps	(Size of a page - size of a PEB) / 4
ProcessHeaps	First byte after PEB
OSMajorVersion	*NtMajorVersion* kernel variable
OSMinorVersion	*NtMinorVersion* kernel variable
OSBuildNumber	*NtBuildNumber* kernel variable & 0x3FFF
OSPlatformId	2

If the image file specifies explicit Windows version values, this information replaces the initial values shown in Table 6-6. The mapping from image version information fields to PEB fields is described in Table 6-7.

Table 6-7 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
OSPlatformId	(OptionalHeader.Win32VersionValue >> 30) ^ 0x2

Stage 2F: Completing the Setup of the Executive Process Object

Before the handle to the new process can be returned, a few final setup steps must be completed:

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 new process is added to the job. (Jobs are described at the end of this chapter.)
3. If the image header characteristic's 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. This choosing process is done 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 out across the processors evenly.
4. 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.
5. *CreateProcess* inserts the new process block at the end of the Windows list of active processes (*PsActiveProcessHead*).
6. The process's creation time is set, the handle to the new process is returned to the caller (*CreateProcess* in Kernel32.dll).

Stage 3: 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. Before the thread can be created, it needs a stack and a context in which to run, so these are set up now. The stack size for the initial thread is taken from the image—there's no way to specify another size.

Now the initial thread can be created, which is done by calling *NtCreateThread*. 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). 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). *NtCreateThread* calls *PspCreateThread* (a function also used to create system threads) and performs the following steps:

1. The thread count in the process object is incremented.
2. An executive thread block (ETHREAD) is created and initialized.
3. A thread ID is generated for the new thread.
4. The TEB is set up in the user-mode address space of the process.
5. The user-mode thread start address is stored in the ETHREAD. For Windows threads, this is the system-supplied thread startup function in Kernel32.dll (*BaseProcessStart* for the first thread in a process and *BaseThreadStart* for additional threads). The user's specified Windows start address is stored in the ETHREAD block in a different location so that the system-supplied thread startup function can call the user-specified startup function.

6. *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 *PspCreateThread*.
7. Any registered systemwide thread creation notification routines are called.
8. The thread's access token is set to point to the process access token, and an access check is made to determine whether the caller has the right to create the thread. This check will always succeed if you're creating a thread in the local process, but it might fail if you're using *CreateRemoteThread* to create a thread in another process and the process creating the thread doesn't have the debug privilege enabled.
9. Finally, the thread is readied for execution.

Stage 4: Notifying the Windows Subsystem about the New Process

If software restriction policies dictate, a restricted token is created for the new process. At this point, all the necessary executive process and thread objects have been created. Kernel32.dll next sends a message to the Windows subsystem so that it can set up for the new process and 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)

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

1. *CreateProcess* 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, *CreateProcess* 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 an exception occurs in the process. (For further information on exception handling, see Chapter 3.)

5. If the process is being debugged (that is, if it is attached to a debugger process), the process debug port is set to the Windows subsystem's general function port. This setting ensures that Windows will send debug events that occur in the new process (such as thread creation and deletion, exceptions, and so on) as messages to the Windows subsystem so that it can then dispatch the events to the process that is acting as the new process's debugger.
6. The Csrss thread block is allocated and initialized.
7. *CreateProcess* inserts the thread in the list of threads for the process.
8. The count of processes in this session is incremented.
9. The process shutdown level is set to 0x280 (the default process shutdown level—see *SetProcessShutdownParameters* in the MSDN Library documentation for more information).
10. The new process block is inserted into the list of Windows subsystemwide processes.
11. The per-process data structure used by the kernel-mode part of the Windows subsystem (W32PROCESS structure) is allocated and initialized.
12. The application start cursor is displayed. This cursor is the familiar arrow with an hourglass attached—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, *CreateProcess* waits 5 seconds for the application to show a window. After that time, *CreateProcess* will reset the cursor again.

Stage 5: 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 6).

Stage 6: 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.

On Windows 2000, *PspUserThreadStartup* first enables working set expansion. If the process being created is a debuggee, all threads in the process are suspended. (Threads might have been created during Stage 3.) A create process message is then sent to the process's debug port (which is the Windows subsystem function port, because this is a Windows process) so that the subsystem can deliver the process startup debug event (CREATE_PROCESS_DEBUG_INFO) to the appropriate debugger process. *PspUserThreadStartup* then waits for the Windows subsystem to get the reply from the debugger (via the *ContinueDebugEvent* function). When the Windows subsystem replies, all the threads are resumed.

On Windows XP and Windows Server 2003, *PspUserThreadStartup* checks whether application prefetching is enabled on the system and, if so, calls the logical prefetcher to process the prefetch instruction file (if it exists) and prefetch pages referenced during the first 10 seconds the process started last time. (For details on the prefetcher, see Chapter 3.) Finally, *PspUserThreadStartup* queues a user-mode APC to run the image loader initialization routine (*LdrInitializeThunk* in Ntdll.dll). The APC will be delivered when the thread attempts to return to user mode.

When *PspUserThreadStartup* returns to *KiThreadStartup*, it returns from kernel mode, the APC is delivered, and *LdrInitializeThunk* is called. The *LdrInitializeThunk* routine initializes the loader, heap manager, NLS tables, thread-local storage (TLS) 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 introduced in Windows XP to address DLL versioning problems.)

Finally, the image begins execution in user mode when the loader initialization returns to the user mode APC dispatcher, which then calls the thread’s start function that was pushed on the user stack when the user APC was delivered.

Side-by-Side Assemblies

A problem that has long plagued Windows users is “DLL hell.” You enter DLL hell when you install an application that replaces one or more core system DLLs, such as those for common controls, the Microsoft Visual Basic runtime, or MFC. Application installation programs make these replacements to ensure that the application runs properly, but at the same time, updated DLLs might have incompatibilities with other already-installed applications.

Windows 2000 partly addressed DLL hell by preventing the modification of core system DLLs with the Windows File Protection feature, and by allowing 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. This type of DLL redirection avoids application/DLL incompatibility problems, but it does so at the expense of sharing DLLs, which is one of the points of DLLs in the first place. In addition, 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 XP introduces 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, which is new to Windows XP. Its manifest file is named \Windows\Winsxs\Manifest\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 includes integration with Windows XP themes, and because applications not written with themes-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.

You can see the effect of a manifest that directs the system to use the new common control library on Windows XP by running the User State Migration Wizard (\Windows\System32\Usmt\Migwiz.exe) with and without its manifest file:

1. Run it, and notice the Windows XP themes on the buttons in the wizard.
2. Open the Migwiz.exe.manifest file in Notepad, and locate the inclusion of the version 6 common control library.
3. Rename the Migwiz.exe.manifest to **Migwiz.exe.manifest.bak**.
4. Rerun the wizard, and notice the unthemed buttons.
5. Restore the manifest file to its original name.

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.

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 6-7. 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. In addition, the Windows subsystem process (Csrss) maintains a parallel structure for each thread created in a Windows process. 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.

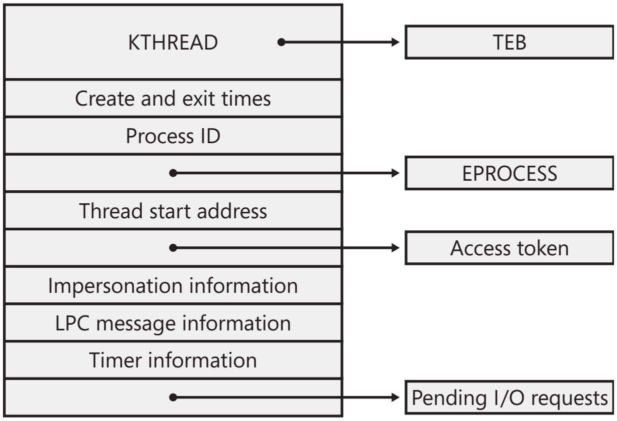


Figure 6-7 Structure of the executive thread block

Most of the fields illustrated in Figure 6-7 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 LPC messages and pending I/O requests. As you can see in Table 6-8, 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 6-8 **Key Contents of the Executive Thread Block**

Element	Description	Additional Reference
KTHREAD	See Table 6-9.	
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 8
LPC information	Message ID that the thread is waiting for and address of message.	Local procedure calls (Chapter 3)
I/O information	List of pending I/O request packets (IRPs).	I/O system (Chapter 9)

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 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 6-8.

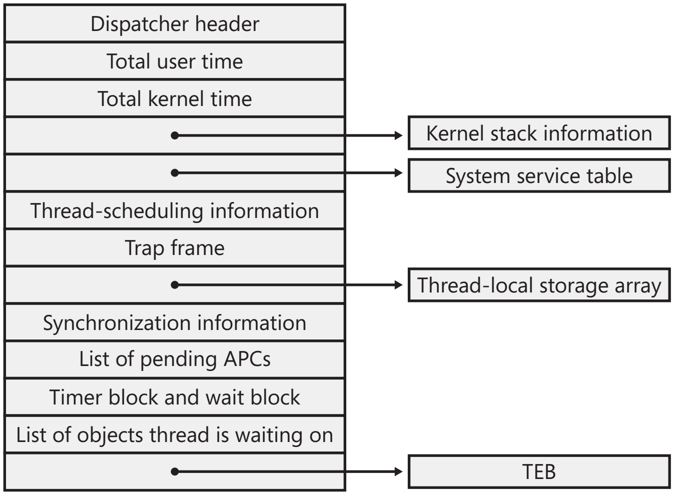

Dispatcher header
Total user time
Total kernel time
Kernel stack information
System service table
Thread-scheduling information
Trap frame
Thread-local storage array
Synchronization information
List of pending APCs
Timer block and wait block
List of objects thread is waiting on
TEB

Figure 6-8 Structure of the kernel thread block

The key fields of the KTHREAD block are described briefly in Table 6-9.

Table 6-9 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.	
Pointer to kernel stack information	Base and upper address of the kernel stack.	Memory management (Chapter 7)
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)
Scheduling information	Base and current priority, quantum, affinity mask, ideal processor, scheduling state, freeze count, and suspend count.	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)

Table 6-9 Key Contents of the KTHREAD Block

Element	Description	Additional Reference
Wait information	List of objects the thread is waiting for, wait reason, 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, and alertable flag.	Aynchronous Procedure Call (APC) Interrrupts (Chapter 3)
Timer block	Built-in timer block (also a corresponding wait block).	
Queue list	Pointer to queue object that the thread is associated with.	Synchronization (Chapter 3)
Pointer to TEB	Thread ID, TLS information, PEB pointer, and GDI and OpenGL 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:

```
lkd> dt nt!_ethread
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
   +0x1dc KeyedWaitValue   : Ptr32 Void
   +0x1e0 ActiveTimerListLock : Uint4B
   +0x1e4 ActiveTimerListHead : _LIST_ENTRY
   +0x1ec Cid              : _CLIENT_ID
   +0x1f4 LpcReplySemaphore : _KSEMAPHORE
   +0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
   +0x208 LpcReplyMessage  : Ptr32 Void
   +0x208 LpcWaitingOnPort : Ptr32 Void
   +0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
   +0x210 IrpList          : _LIST_ENTRY
   +0x218 TopLevelIrp      : Uint4B
   +0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
   +0x220 ThreadsProcess   : Ptr32 _EPROCESS
   +0x224 StartAddress     : Ptr32 Void
   +0x228 Win32StartAddress : Ptr32 Void
   +0x228 LpcReceivedMessageId : Uint4B
   +0x22c ThreadListEntry  : _LIST_ENTRY
   +0x234 RundownProtect   : _EX_RUNDOWN_REF
   +0x238 ThreadLock       : _EX_PUSH_LOCK
   +0x23c LpcReplyMessageId : Uint4B
   +0x240 ReadClusterSize  : Uint4B
   +0x244 GrantedAccess    : Uint4B
   +0x248 CrossThreadFlags : Uint4B
   +0x248 Terminated       : Pos 0, 1 Bit
   +0x248 DeadThread       : Pos 1, 1 Bit
   +0x248 HideFromDebugger : Pos 2, 1 Bit
   +0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
   +0x248 SystemThread     : Pos 4, 1 Bit
   +0x248 HardErrorsAreDisabled : Pos 5, 1 Bit
   +0x248 BreakOnTermination : Pos 6, 1 Bit
   +0x248 SkipCreationMsg  : Pos 7, 1 Bit
   +0x248 SkipTerminationMsg : Pos 8, 1 Bit
   +0x24c SameThreadPassiveFlags : Uint4B
   +0x24c ActiveExWorker   : Pos 0, 1 Bit
   +0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
   +0x24c MemoryMaker      : Pos 2, 1 Bit
   +0x250 SameThreadApcFlags : Uint4B
   +0x250 LpcReceivedMsgIdValid : Pos 0, 1 Bit
   +0x250 LpcExitThreadCalled : Pos 1, 1 Bit
   +0x250 AddressSpaceOwner : Pos 2, 1 Bit
   +0x254 ForwardClusterOnly : UChar
   +0x255 DisablePageFaultClustering : UChar
```

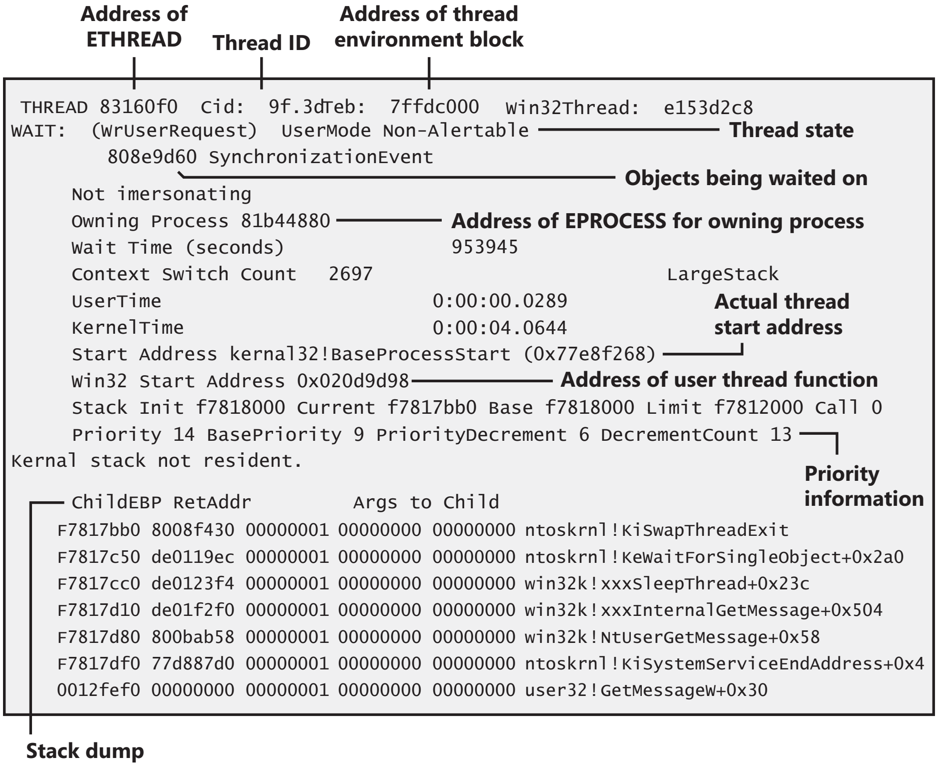
The KTHREAD can be displayed with a similar command:

```
lkd> dt nt!_kthread
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY
   +0x018 InitialStack     : Ptr32 Void
   +0x01c StackLimit       : Ptr32 Void
   +0x020 Teb              : Ptr32 Void
   +0x024 TlsArray         : Ptr32 Void
   +0x028 KernelStack      : Ptr32 Void
   +0x02c DebugActive      : UChar
   +0x02d State            : UChar
   +0x02e Alerted          : [2] UChar
   +0x030 Iopl             : UChar
   +0x031 NpxState         : UChar
   +0x032 Saturation       : Char
   +0x033 Priority         : Char
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : Uint4B
   +0x050 IdleSwapBlock    : UChar
   +0x051 Spare0           : [3] UChar
   +0x054 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 Windows Debugging Tools. Notice that the thread list shows the "Win32StartAddress." 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 Kernel32.dll), not the application-specified start address.

```
C:\> tlist winword
 155 WINWORD.EXE       Document1 - Microsoft Word
   CWD:     C:\book\
   CmdLine: "C:\Program Files\Microsoft Office\Office\WINWORD.EXE"
   VirtualSize:    64448 KB   PeakVirtualSize:   106748 KB
   WorkingSetSize:  1104 KB   PeakWorkingSetSize:  6776 KB
   NumberOfThreads: 2
    156 Win32StartAddr:0x5032cfdb LastErr:0x00000000 State:Waiting
    167 Win32StartAddr:0x00022982 LastErr:0x00000000 State:Waiting
                     0x50000000  WINWORD.EXE
     5.0.2163.1 shp  0x77f60000  ntdll.dll
     5.0.2191.1 shp  0x77f00000  KERNEL32.dll
         §          list of DLLs loaded in process
```

The TEB, illustrated in Figure 6-9, 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.

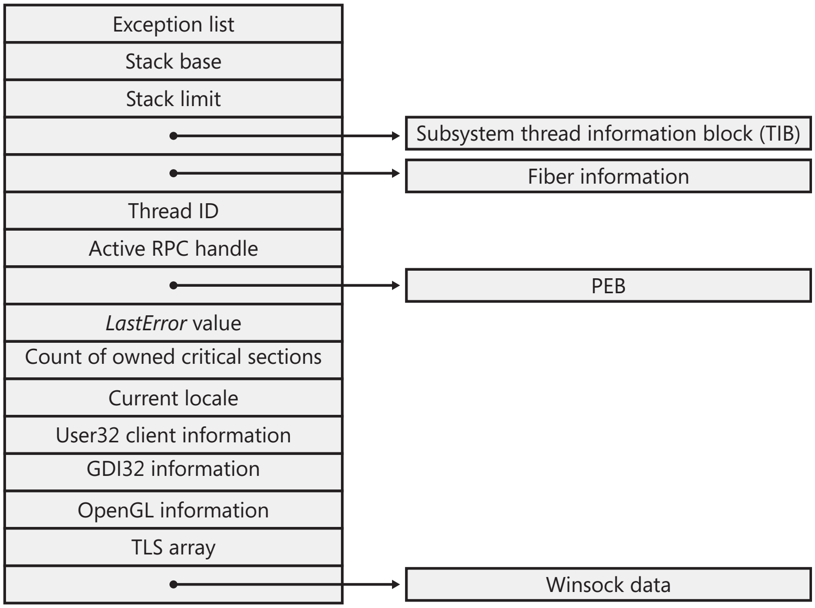


Figure 6-9 Fields of the thread environment block

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:        0006b540
    StackBase:            00070000
    StackLimit:           00065000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffde000
    EnvironmentPointer:   00000000
    ClientId:             00000254 . 000007ac
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdf000
    LastErrorValue:       2
    LastStatusValue:      c0000034
    Count Owned Locks:    0     HardErrorMode:        0
```

Kernel Variables

As with processes, a number of Windows kernel variables control how threads run. Table 6-10 shows the kernel-mode kernel variables that relate to threads.

Table 6-10 Thread-Related Kernel Variables

Variable	Type	Description
PspCreateThreadNotifyRoutine	Array of pointers	Array of pointers to routines to be called on during thread creation and deletion (maximum of eight).
PspCreateThreadNotifyRoutineCount	DWORD	Count of registered thread-notification routines.
PspCreateProcessNotifyRoutine	Array of pointers	Array of pointers to routines to be called on during process creation and deletion (maximum of eight).

Performance Counters

Most of the key information in the thread data structures is exported as performance counters, which are listed in Table 6-11. You can extract much information about the internals of a thread just by using the Performance tool in Windows.

Table 6-11 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. This ID is valid only during the process's lifetime because process IDs are re-used.
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.)
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 6-12 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 6-12 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
GetExitCodeThread	Gets another thread's exit code
GetThreadTimes	Returns timing information for a thread
GetCurrentProcess	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* creates a user-mode stack for the thread in the process's address space.
2. *CreateThread* initializes the thread's hardware context (CPU architecture–specific). (For further information on the thread context block, see the Windows API reference documentation on the CONTEXT structure.)
3. *NtCreateThread* is called to create the executive thread object in the suspended state. For a description of the steps performed by this function, see the description of Stage 3 and Stage 6 in the section "Flow of *CreateProcess*."
4. *CreateThread* notifies the Windows subsystem about the new thread, and the subsystem does some setup work for the new thread.
5. The thread handle and the thread ID (generated during step 3) are returned to the caller.

6. 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 6: Performing Process Initialization in the Context of the New Process” before calling the actual user’s specified start address.

Examining Thread Activity

Besides the Performance tool, several other tools expose various elements of the state of Windows threads. (The tools that show thread-scheduling information are listed in the section “Thread Scheduling.”) These tools are itemized in Figure 6-10.

> **Note** To display thread details with Tlist, you must type **tlist xxx**, where *xxx* is a process image name or window title. (Wildcards are supported.)

Object	Perfmon	Pviewer	Pstat	Qslice	Tlist	KD *!thread*	Process Explorer	Pslist
Thread ID	✓	✓	✓		✓	✓	✓	✓
Actual start address	✓	✓	✓			✓	✓	✓
Win32 start address					✓	✓	✓	
Current address	✓	✓				✓	✓	
Number of context switches	✓	✓	✓				✓	✓
Total user time		✓	✓			✓	✓	✓
Total privileged time		✓	✓			✓	✓	✓
Elapsed time	✓	✓				✓	✓	✓
Thread state	✓		✓		✓	✓	✓	✓
Reason for wait state	✓		✓		✓	✓	✓	✓
Last error					✓		✓	
Percentage of CPU time	✓			✓			✓	
Percentage of user time	✓			✓			✓	
Percentage of privileged time	✓			✓			✓	
Address of TEB						✓		
Address of ETHREAD						✓		
Objects waiting on						✓		

Figure 6-10 Thread-related tools and their functions

Process Explorer provides easy access to thread activity within a process. This is especially important if you are trying to determine why a process is running that is hosting multiple services (such as Svchost.exe, Dllhost.exe, Inetinfo.exe, or the System process) or why a process is hung.

To view the threads in a process, 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. 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 context switch delta represents the number of times that thread began running in between the refreshes configured for Process Explorer. It provides a different way to determine thread activity than using the percentage of CPU consumed. In some ways it is better because many threads run for such a short amount of time that they are seldom (if ever) the currently running thread when the interval clock timer interrupt occurs, and therefore, are not charged for their CPU time. For example, if you add the context switch delta column to the process display and sort by that column, you will see processes that have threads running but that also have a CPU time percentage of zero (or very small).

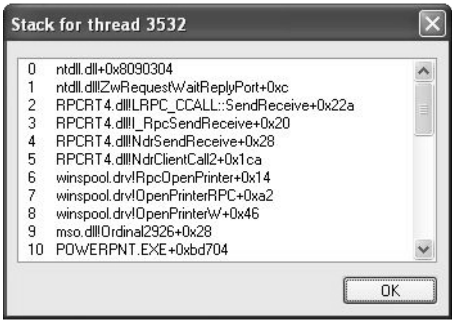
The thread start address is displayed in the form "*module!function*", where *module* is the name of the .exe or .dll. The function name relies on access to symbol files for the module. (See "Experiment: Viewing Process Details with Process Explorer" in Chapter 1.) If you are unsure what the module is, press 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 process or thread startup wrapper function (*BaseProcessStart* or *BaseThreadStart* in Kernel32.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, 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 Livekd from *www.sysinternals.com*. However, it is more difficult to use. Note that running Windbg in local kernel debugging mode, which is supported only on Windows XP or Windows Server 2003, does not show thread stacks.

Viewing the thread stack can also help you determine why a process is hung. As an example, on one system, Microsoft 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 6-11.



```
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 6-11 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.

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 6-13 (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, two threads are ready to run at priority 10 and six at priority 8. Because this output was generated using LiveKd on a uniprocessor system, the current thread will always be the kernel debugger (Kd or WinDbg).

```
kd> !ready 1
Ready Threads at priority 10
 THREAD 810de030  Cid 490.4a8 Teb: 7ffd9000  Win32Thread: e297e008 READY
 THREAD 81110030  Cid 490.48c Teb: 7ffde000  Win32Thread: e29425a8 READY
Ready Threads at priority 8
 THREAD 811fe790  Cid 23c.274 Teb: 7ffdb000  Win32Thread: e258cda8 READY
 THREAD 810bec70  Cid 23c.50c Teb: 7ffd9000  Win32Thread: e2ccf748 READY
 THREAD 8003a950  Cid 23c.550 Teb: 7ffda000  Win32Thread: e29a7ae8 READY
 THREAD 85ac2db0  Cid 23c.5e4 Teb: 7ffd8000  Win32Thread: e297a9e8 READY
 THREAD 827318d0  Cid 514.560 Teb: 7ffd9000  Win32Thread: 00000000 READY
 THREAD 8117adb0  Cid 2d4.338 Teb: 7ffaf000  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 6-12, 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

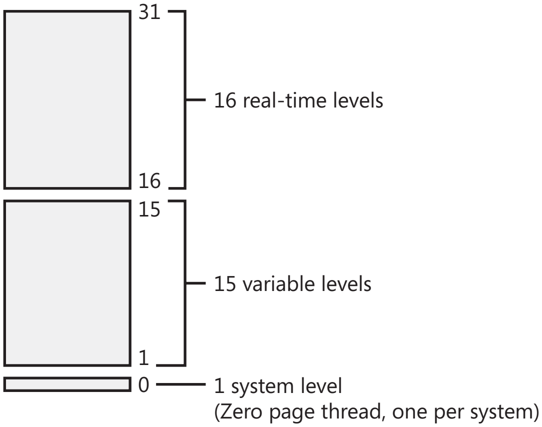


Figure 6-12 Thread priority levels

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 6-13.

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.

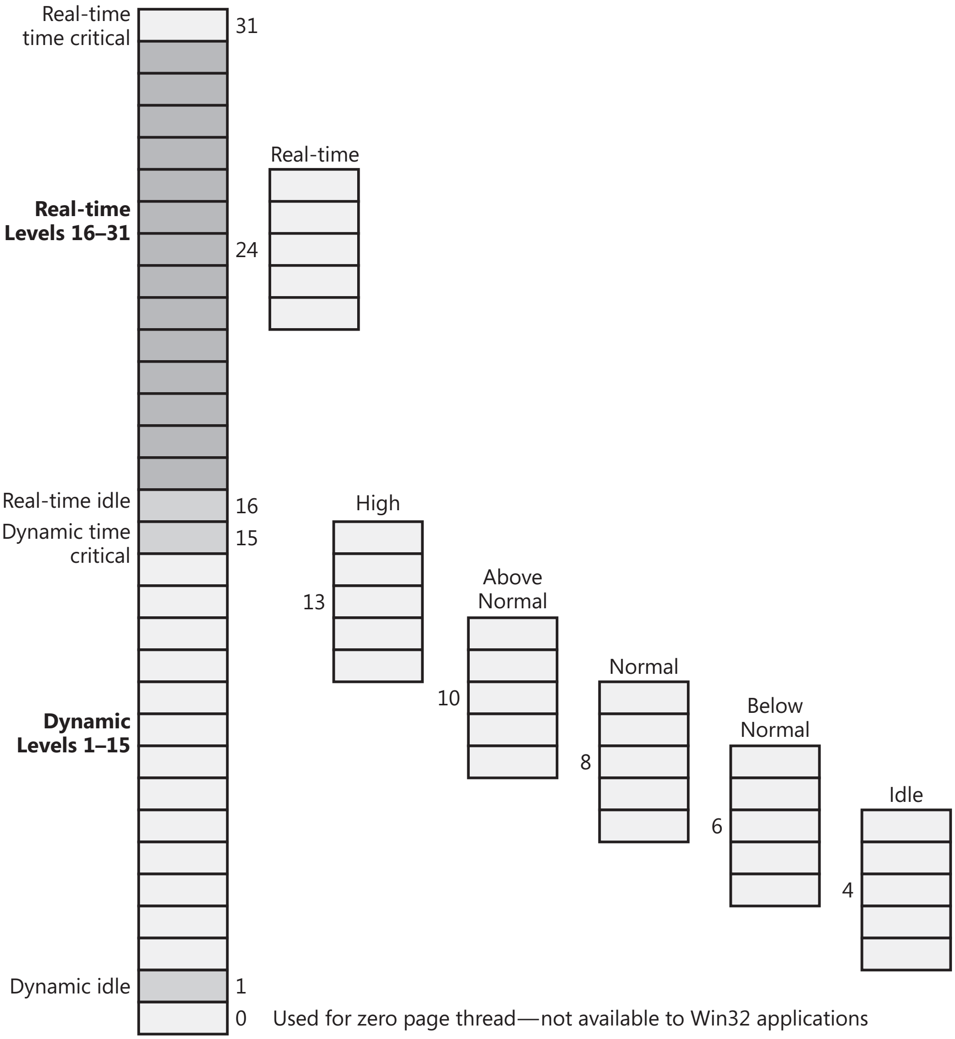


Figure 6-13 Mapping of Windows kernel priorities to the Windows API

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 its 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 6-13. (For more information, see the Windows API reference documentation.)

Table 6-13 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.
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.)
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

The following table lists the tools related to thread scheduling. You can change (and view) the base process priority with a number of different tools, such as Task Manager, Process Explorer, Pview, or Pviewer. Note that you can kill individual threads in a process with Process Explorer. This should be done, of course, with extreme care.

You can view individual thread priorities with the Performance tool, Process Explorer, Pslist, Pview, Pviewer, and Pstat. 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 would understand the relative importance of the threads within the process.

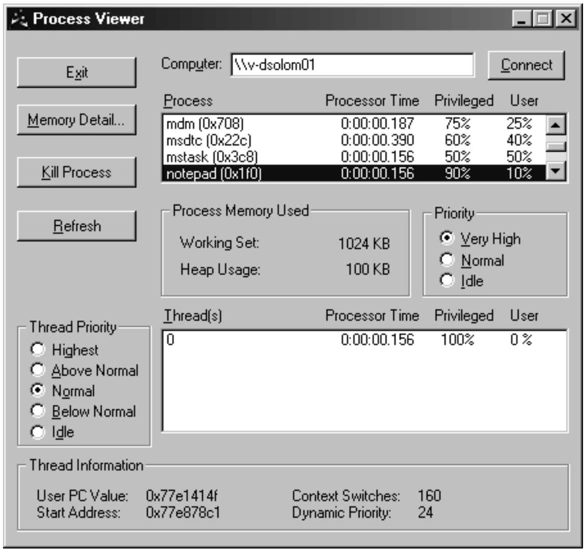
						KD	**Process**
Object	**Taskman**	**Perfmon**	**Pviewer**	**Pview**	**Pstat**	***!thread***	**Explorer**
Process priority class	✓		✓	✓			✓
Process base priority		✓			✓		✓
Thread base priority		✓					✓
Thread current priority		✓	✓	✓	✓	✓	✓

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 the command prompt, type **start /realtime notepad**. Notepad should open.
2. Run either Process Explorer or the Process Viewer utility in the Support Tools (Pviewer.exe), and select Notepad.exe from the list of processes, as shown here. Notice that the dynamic priority of the thread in Notepad is 24. This matches the real-time value shown in Figure 6-13.



Computer: \\v-dsolom01

Process	Processor Time	Privileged	User
mdm (0x708)	0:00:00.187	75%	25%
msdtc (0x22c)	0:00:00.390	60%	40%
mstask (0x3c8)	0:00:00.156	50%	50%
notepad (0x1f0)	0:00:00.156	90%	10%

Process Memory Used

Working Set: 1024 KB

Heap Usage: 100 KB

Priority

● Very High
○ Normal
○ Idle

Thread(s)	Processor Time	Privileged	User
0	0:00:00.156	100%	0 %

Thread Priority

○ Highest
○ Above Normal
● Normal
○ Below Normal
○ Idle

Thread Information

User PC Value: 0x77e1414f

Start Address: 0x77e878c1

Context Switches: 160

Dynamic Priority: 24

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.

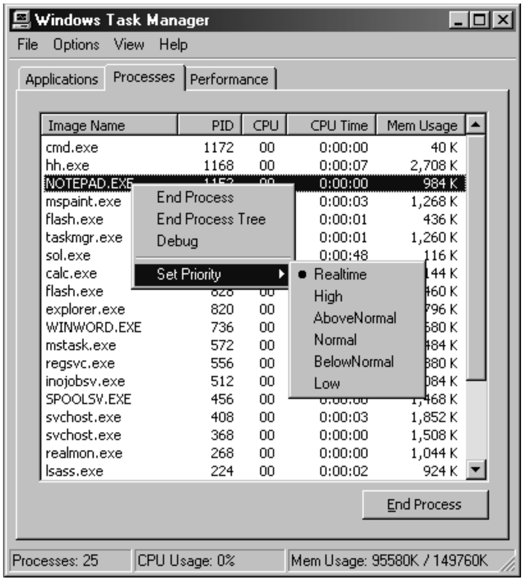


Image Name	PID	CPU	CPU Time	Mem Usage
cmd.exe	1172	00	0:00:00	40 K
hh.exe	1168	00	0:00:07	2,708 K
NOTEPAD.EXE	1152	00	0:00:00	984 K
mspaint.exe			0:00:03	1,268 K
flash.exe			0:00:01	436 K
taskmgr.exe			0:00:01	1,260 K
sol.exe			0:00:48	116 K
calc.exe				144 K
flash.exe	828	00		460 K
explorer.exe	820	00		796 K
WINWORD.EXE	736	00		680 K
mstask.exe	572	00		484 K
regsvc.exe	556	00		880 K
inojobsv.exe	512	00		084 K
SPOOLSV.EXE	456	00	0:00:00	1,468 K
svchost.exe	408	00	0:00:03	1,852 K
svchost.exe	368	00	0:00:00	1,508 K
realmon.exe	268	00	0:00:00	1,044 K
lsass.exe	224	00	0:00:02	924 K

Processes: 25 CPU Usage: 0% Mem Usage: 95580K / 149760K

Windows System Resource Manager

Windows Server 2003, Enterprise Edition and Windows Server 2003, 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 7 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).

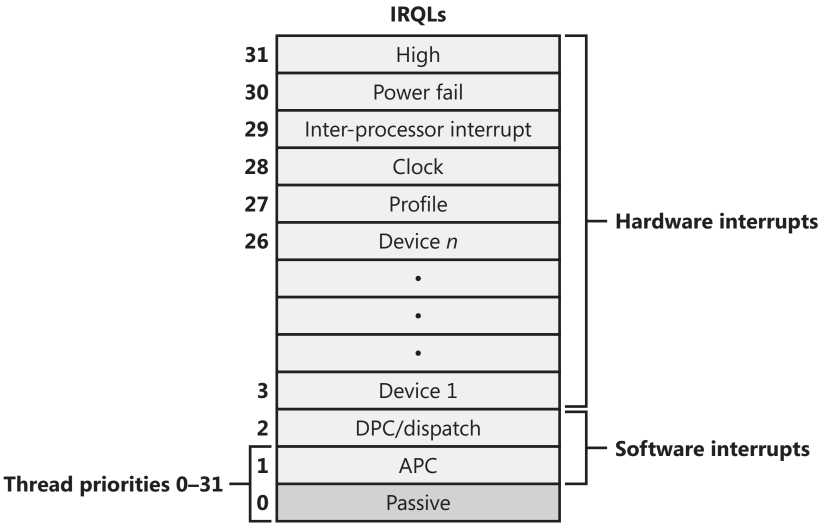
Real-Time Priorities

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, threads normally run at IRQL 0 or 1. (For a description of how Windows uses interrupt levels, see Chapter 3.) User-mode threads always run 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 6-14 illustrates the state transitions for threads on Windows 2000 and Windows XP. (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.

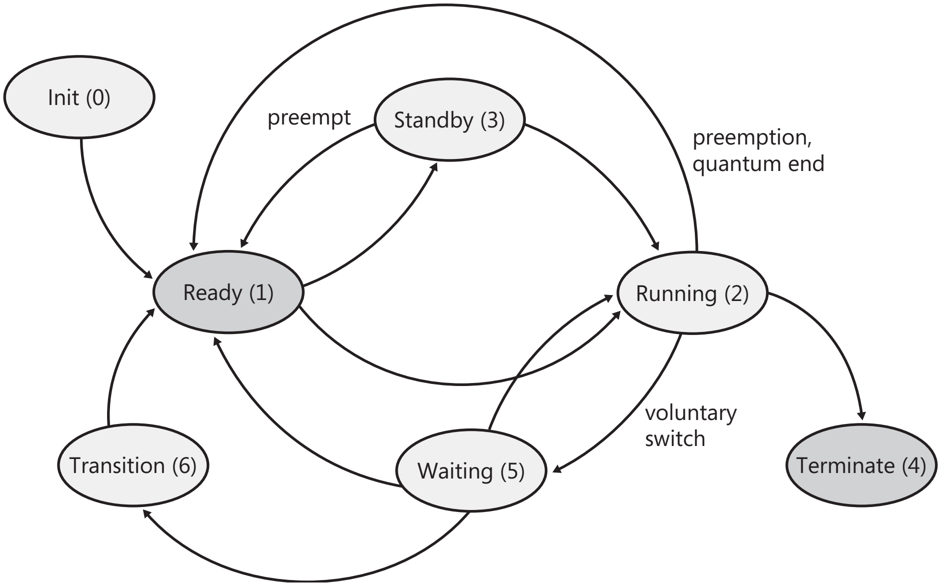


Figure 6-14 Thread states on Windows 2000 and Windows XP

The thread states are as follows:

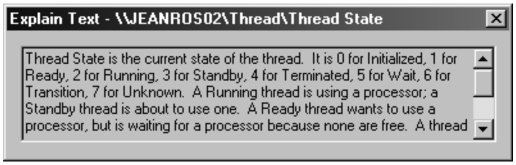
- **Ready** A thread in the ready state is waiting to execute. When looking for a thread to execute, the dispatcher considers only the pool of threads in the ready state.
- **Standby** A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exist, 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.

- **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.

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 if 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:

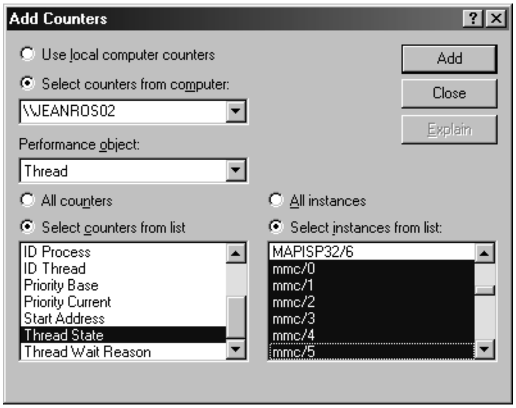
1. Run the Microsoft Notepad utility (Notepad.exe).
2. Start the Performance tool by selecting Programs from the Start menu and then selecting Performance from the Adminstrative Tools menu.
3. Select chart view if you're in some other view.
4. Right-click on the graph, and choose Properties.
5. Click 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. Click the Explain button to see the definition of the values:



Explain Text - \\JEANROS02\Thread\Thread State

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

8. In the Instances box, 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.



Add Counters

○ Use local computer counters

● Select counters from computer:

\\JEANROS02

Performance object:

Thread

○ All counters

● Select counters from list

ID Process
ID Thread
Priority Base
Priority Current
Start Address
Thread State
Thread Wait Reason

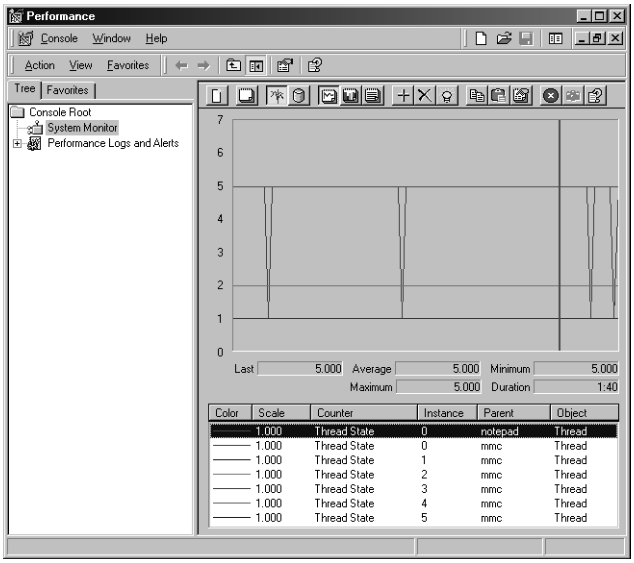
○ All instances

● Select instances from list:

MAPISP32/6
mmc/0
mmc/1
mmc/2
mmc/3
mmc/4
mmc/5

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

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 5, 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

Color	Scale	Counter	Instance	Parent	Object
———	1.000	Thread State	0	notepad	Thread
———	1.000	Thread State	0	mmc	Thread
———	1.000	Thread State	1	mmc	Thread
———	1.000	Thread State	2	mmc	Thread
———	1.000	Thread State	3	mmc	Thread
———	1.000	Thread State	4	mmc	Thread
———	1.000	Thread State	5	mmc	Thread

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.

The state diagram for threads on Windows Server 2003 is shown in Figure 6-15. Notice the new state called *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 new state exists so that the kernel can minimize the amount of time the systemwide lock on the scheduling database is held. (This process is explained further in the section "Multiprocessor Dispatcher Database.")

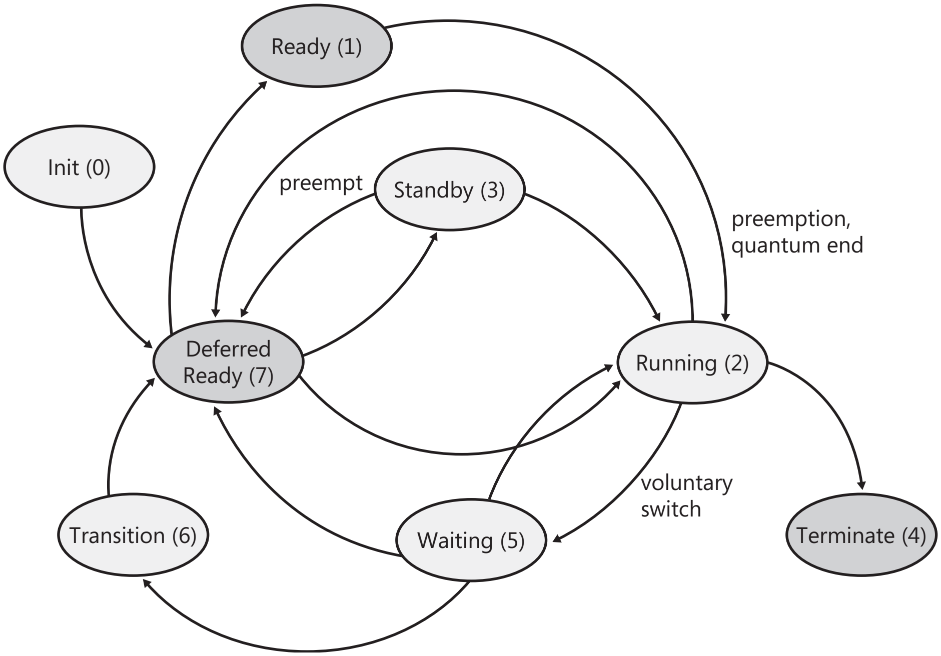


Figure 6-15 Thread states on Windows Server 2003

Dispatcher Database

To make thread-scheduling decisions, the kernel maintains a set of data structures known collectively as the *dispatcher database*, illustrated in Figure 6-16. The dispatcher database keeps track of which threads are waiting to execute and which processors are executing which threads.

Note The dispatcher database on a uniprocessor system has the same structure as on multiprocessor Windows 2000 and Windows XP systems, but is different on Windows Server 2003 systems. These differences, as well as the differences in the way Windows selects threads to run on multiprocessor systems, are explained in the section "Multiprocessor Systems."

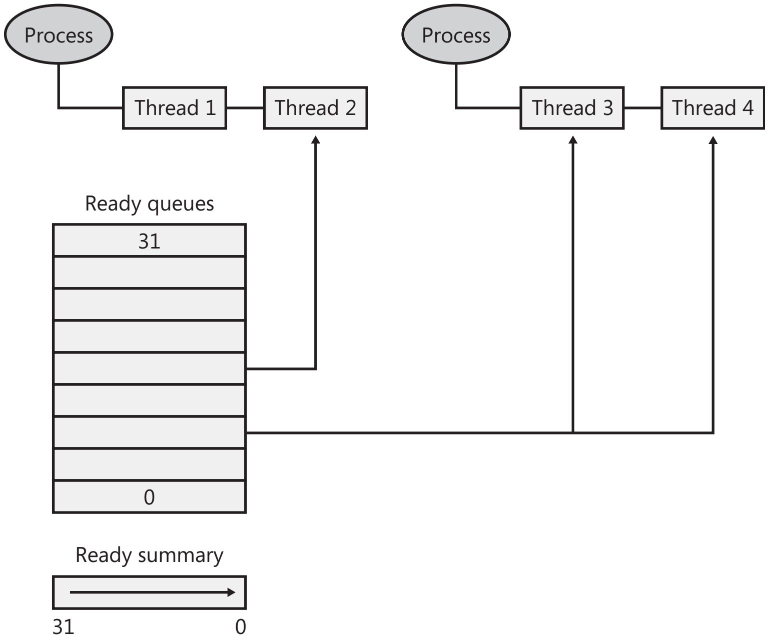


Figure 6-16 Dispatcher database (uniprocessor and Windows 2000/XP multiprocessor)

The dispatcher *ready queues (KiDispatcherReadyListHead)* 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* (*KiReadySummary*). Each bit set indicates one or more threads in the ready queue for that priority level. (Bit 0 represents priority 0, and so on.)

Table 6-15 lists the kernel-mode kernel variables that are related to thread scheduling on uniprocessor systems.

Table 6-14 Thread-Scheduling Kernel Variables

Variable	Type	Description
KiReadySummary	Bitmask (32 bits)	Bitmask of priority levels that have one or more ready threads
KiDispatcherReadyListHead	Array of 32 list entries	List heads for the 32 ready queues

On uniprocessor systems, the dispatcher database is synchronized by raising IRQL to DPC/dispatch level and SYNCH_LEVEL (both of which are 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 because threads normally run at IRQL 0 or 1. On multiprocessor systems, more is required than raising IRQL because each processor can, at the same time, raise to the same IRQL and attempt to operate on the dispatcher database. How Windows synchronizes access to the dispatcher database on multiprocessor systems 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 2000 Professional and Windows XP, 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 multiprocessors it is about 15 milliseconds. (The actual clock rate is not exactly a round number of milliseconds—see the following experiment for a way to check the actual clock interval.)

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 *www.sysinternals.com*. Here’s the output from a uniprocessor x86 system:

```
C:\>clockres

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

The system clock interval is 10.014400 ms
```

Quantum Accounting

Each process has a quantum value in the kernel process block. This value is used when giving a thread a new quantum. As a thread runs, its quantum is reduced at each clock interval. If there is no remaining thread quantum, the 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. Note that when the clock interrupt interrupts a DPC or another interrupt that was in progress, the thread that was in the running state has its quantum deducted, even if it hadn't been running for a full clock interval. If this was not done and device interrupts or DPCs occurred right before the clock interval timer interrupts, threads might not ever get their quantum reduced.

Internally, the quantum value is stored as a multiple of three times the number of clock ticks. This means that on Windows 2000 and Windows XP systems, threads, by default, have a quantum value of 6 (2 * 3), and Windows Server systems have a quantum value of 36 (12 * 3). Each time the clock interrupts, the clock-interrupt routine deducts a fixed value (3) from the thread quantum.

The reason quantum is stored internally in terms of a multiple of 3 quantum units per clock tick rather than as single units is to allow for partial quantum decay on wait completion. When a thread that has a current priority less than 16 and a base priority less than 14 executes a wait function (such as *WaitForSingleObject* or *WaitForMultipleObjects*) that is satisfied immediately (for example, without having to wait), its quantum is reduced by 1 quantum unit. In this way, threads that wait will eventually expire their quantum.

In the case where a wait is not satisfied immediately, threads below priority 16 also have their quantum reduced by 1 unit (except if the thread is waking up to execute a kernel APC because the wait code will charge the quantum after the wait is actually satisfied). However, before doing the reduction, if the thread is at priority 14 or above, its quantum is reset to a full turn. This is also done for threads running at less than 14 if they are not running with a special priority boost (such as is done for foreground processes or in the case of CPU starvation) and if they are receiving a priority boost as a result of the unwait operation. (Priority boosting is explained in the next section.)

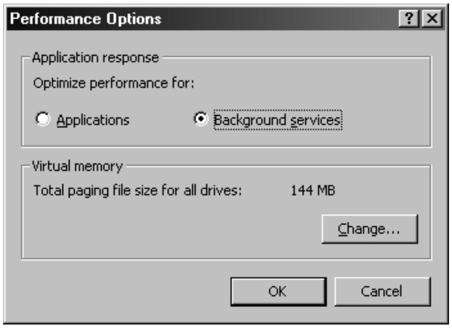
This partial decay addresses the case in which a thread enters a wait state before the clock interval timer fires. If this adjustment were not made, it would be possible for threads never to have their quantums reduced. For example, if a thread ran, entered a wait state, ran again, and entered another 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. Note that this does not apply to zero timeout waits, but only wait operations that do not require waiting because all the wait conditions are fulfilled at the time of the wait.

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 on Windows 2000, right click My Computer, Properties (or go to Control Panel and open the System settings applet), click the Advanced tab, and then click the Performance Options button. The dialog box you will see is shown in Figure 6-17.



Performance Options

Application response

Optimize performance for:

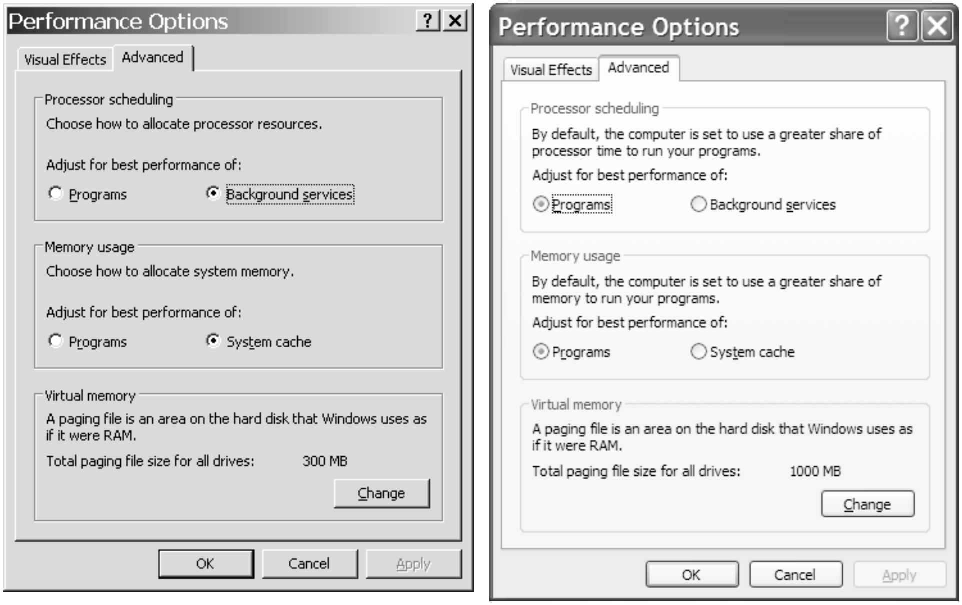
○ Applications ● Background services

Virtual memory

Total paging file size for all drives: 144 MB

Figure 6-17 Quantum Configuration on Windows 2000

To change this on Windows XP and Windows Server 2003, right click My Computer, Properties, click the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. The dialog box displayed is slightly different for Windows XP and Windows Server 2003. These are shown in Figure 6-18.



Performance Options

Processor scheduling

Choose how to allocate processor resources.

Adjust for best performance of:

○ Programs ● Background services

Memory usage

Choose how to allocate system memory.

Adjust for best performance of:

○ Programs ● System cache

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: 300 MB

Performance Options

Processor scheduling

By default, the computer is set to use a greater share of processor time to run your programs.

Adjust for best performance of:

● Programs ○ Background services

Memory usage

By default, the computer is set to use a greater share of memory to run your programs.

Adjust for best performance of:

● Programs ○ System cache

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: 1000 MB

Figure 6-18 Quantum Configuration on Windows XP and Windows Server 2003

The Programs setting (called "Applications" on Windows 2000) designates the use of short, variable quantums—the default for Windows 2000 Professional and Windows XP. If you install Terminal Services on Windows Server 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 systems. The only reason why you might select this 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

Prior to Windows NT 4.0, when a window was brought into the foreground on a workstation or client system, all the threads in the foreground process (the process that owns the thread that owns the window that's in focus) received a priority boost of 2. This priority boost remained in effect while any thread in the process owned the foreground window. The problem with this approach was that if you started a long-running, CPU-intensive process (such as a spreadsheet recalculation) and then switched to another CPU-intensive process (such as a computer-aided design tool, graphics editor, or a game), the process now running in the background would get little or no CPU time because the foreground process would have its threads boosted by 2 (assuming the base priority of the threads in both processes are the same) while it remained in the foreground.

This default behavior was changed as of Windows NT 4.0 Workstation to instead triple the quantum of the threads in the foreground process. Thus, threads in the foreground process run with a quantum of 6 clock ticks, whereas threads in other processes have the default workstation 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 that 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 (or Applications, in Windows 2000) 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 systems).

Quantum Settings Registry Value

The user interface to control quantum settings described earlier modify 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 6-19.

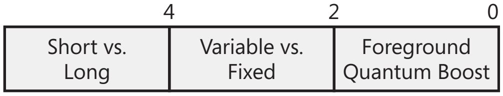


Figure 6-19 Fields of the Win32PrioritySeparation registry value

The fields shown in Figure 6-19 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 2000 Professional and Windows XP, long for Windows Server 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 2000 Professional and Windows XP and fixed for Windows Server systems) will be used.
- **Foreground Quantum Boost** This field (stored in the kernel variable *PsPrioritySeparation*) 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 6-16 shows the possible settings for *PspForegroundQuantum*.

Table 6-15 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.

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 6-20 illustrates a thread entering a wait state and Windows selecting a new thread to run.

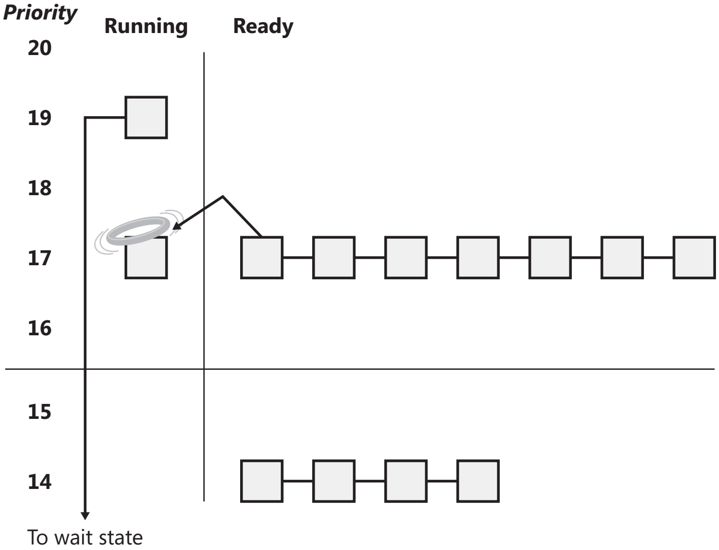


Figure 6-20 Voluntary switching

In Figure 6-20, 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. What about any remaining quantum for the thread? The quantum value isn't reset when a thread enters a wait state—in fact, as explained earlier, when the wait is satisfied, the thread's quantum value is decremented by 1 quantum unit, equivalent to one-third of a clock interval (except for threads running at priority 14 or higher, which have their quantum reset after a wait to a full turn).

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 6-21 illustrates this situation.

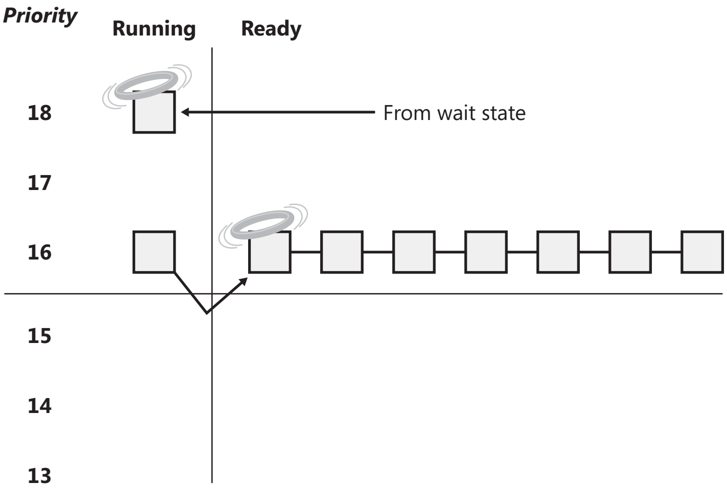


Figure 6-21 Preemptive thread scheduling

In Figure 6-21, a thread with priority 18 emerges from a wait state and repossesses the CPU, causing the thread that had been running (at priority 16) to be bumped to the head of the ready queue. Notice that the bumped thread isn't going to the end of the queue but to the beginning; when the preempting thread has finished running, the bumped thread can complete its quantum.

Quantum End

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

If the thread priority is reduced, Windows looks for a more appropriate thread to schedule. (For example, a more appropriate thread would be a thread in a ready queue with a higher priority than the new priority for the currently running thread.) If the thread priority isn't reduced and there are other threads in the ready queue at the same priority level, Windows selects the next thread in the ready queue at that same priority level and moves the previously running thread to the tail of that queue (giving it a new quantum value and changing its state from running to ready). This case is illustrated in Figure 6-22. If no other thread of the same priority is ready to run, the thread gets to run for another quantum.

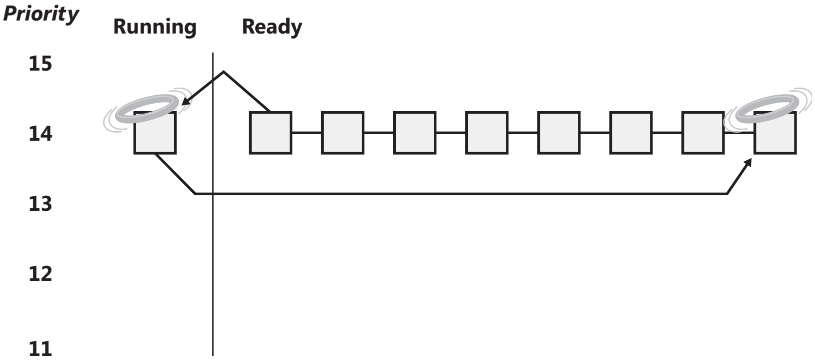


Figure 6-22 Quantum end thread scheduling

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
- User and kernel stack pointers
- 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 7.) 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," Process Viewer reports it as "Idle," Pstat calls it "Idle Process," Process Explode and Tlist call it "System Process," and Qslice calls it "SystemProcess." Windows reports the priority of the idle thread as 0. In reality, however, the idle threads don't have a priority level because they run only when there are no real threads to run. (Remember, only one thread per Windows system is actually running at priority 0—the zero page thread, explained in Chapter 7.)

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.
3. Checks whether a thread has been selected to run next on the processor, and if so, dispatches that thread.
4. Calls the HAL processor idle routine (in case any power management functions need to be performed).

In Windows Server 2003, the idle thread also scans for threads waiting to run on other processors. (This is explained in the upcoming multiprocessor scheduling section.)

Priority Boosts

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

- On completion of I/O operations
- After waiting for executive events or semaphores

- 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.

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 DDK 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 6-17.) 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 6-17, notice that I/O requests to devices that warrant better responsiveness have higher boost values.

Table 6-16 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 base priority, not its current priority. As illustrated in Figure 6-23, 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.

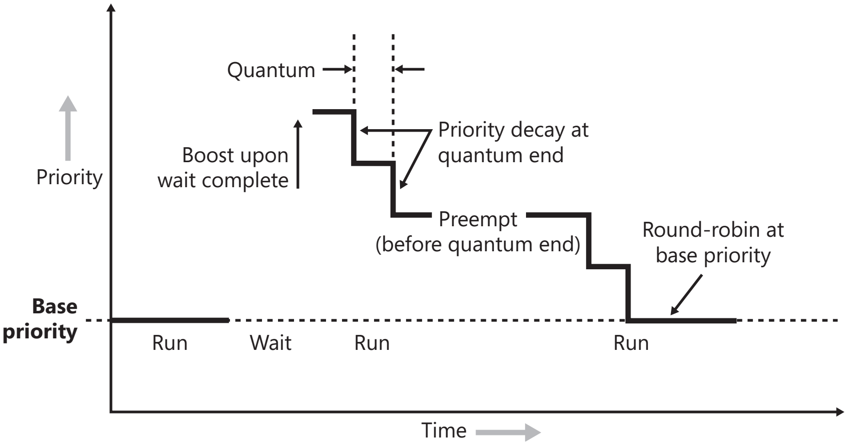

Quantum
Priority decay at quantum end
Boost upon wait complete
Priority
Preempt (before quantum end)
Round-robin at base priority
Base priority
Run
Wait
Run
Run
Time

Figure 6-23 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 DDK 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 over 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 and push locks). 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.

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 *PsPrioritySeparation*. (The windowing system is responsible for determining which process is considered to be in the foreground.) As described in the section on quantum controls, *PsPrioritySeparation* reflects the quantum-table index used to select quantums for the threads of foreground applications. However, in this case, it is being used as a priority boost value.

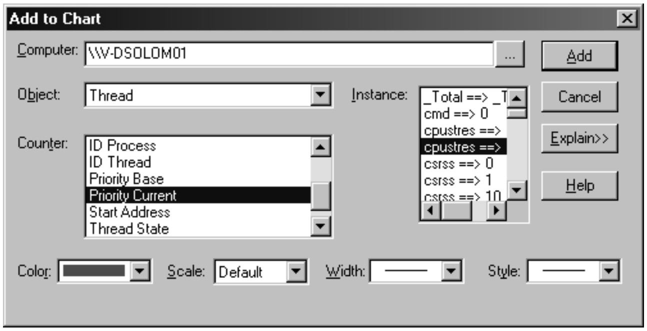
The reason for this boost is to improve the responsiveness of interactive applications—by giving the foreground application a small boost when it completes a wait, it has a better chance of running right away, especially when other processes at the same base priority might be running in the background.

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 (in the resource kit and the Platform SDK), you can watch priority boosts in action. Take the following steps:

1. Open the System utility in Control Panel (or right-click My Computer and select Properties), click the Advanced tab, and click the Performance Options button. Select the Applications option. This causes *PsPrioritySeparation* to get a value of 2.
2. Run Cpustres.exe.
3. Run the Windows NT 4 Performance Monitor (Perfmon4.exe in the Windows 2000 resource kits). This older version of the Performance tool is needed for this experiment because it can query performance counter values at a frequency faster than the Windows Performance tool (which has a maximum interval of once per second).
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add To Chart dialog box.
5. Select the Thread object, and then select the Priority Current counter.
6. In the Instance box, scroll down the list 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 to Chart

Computer: \\V-DSOLOM01

Object: Thread

Counter: ID Process, ID Thread, Priority Base, **Priority Current**, Start Address, Thread State

Instance: _Total ==> _T, cmd ==> 0, cpustres ==>, **cpustres ==>**, csrss ==> 0, csrss ==> 1, csrss ==> 10

Color: ▇ Scale: Default Width: ——— Style: ———

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

8. Select Chart from the Options menu. Change the Vertical Maximum to 16 and the Interval to 0.010, as follows, and click OK:

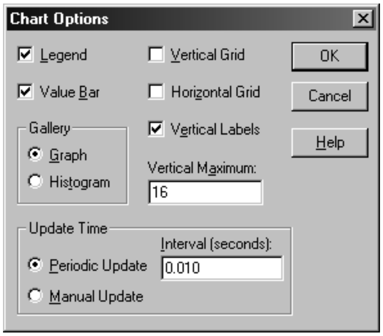


Chart Options

☑ Legend
☐ Vertical Grid
☑ Value Bar
☐ Horizontal Grid
☑ Vertical Labels

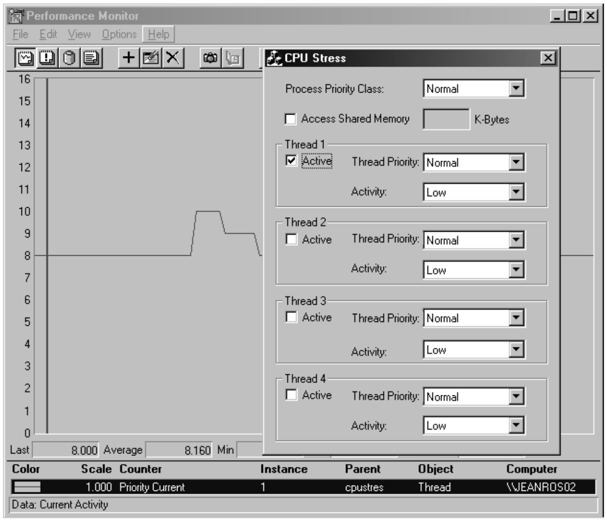
Gallery: ● Graph ○ Histogram

Vertical Maximum: 16

Update Time: ● Periodic Update ○ Manual Update

Interval (seconds): 0.010

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 8.160 Min

Color	Scale	Counter	Instance	Parent	Object	Computer
▭	1.000	Priority Current	1	cpustres	Thread	\\JEANROS02

Data: Current Activity

CPU Stress

Process Priority Class: Normal

☐ Access Shared Memory K-Bytes

Thread 1: ☑ Active Thread Priority: Normal Activity: Low

Thread 2: ☐ Active Thread Priority: Normal Activity: Low

Thread 3: ☐ Active Thread Priority: Normal Activity: Low

Thread 4: ☐ Active Thread Priority: Normal Activity: Low

10. The reason Cpustres receives a boost of 2 periodically is because the thread you're monitoring is sleeping about 75 percent of the time and then waking up—the boost is applied when the thread wakes up. To see the thread get boosted more frequently, increase the Activity level from Low to Medium to Busy. 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 therefore doesn't receive any boosts.
11. When you've finished, exit Performance Monitor and CPU Stress.

Priority Boosts after GUI Threads Wake Up

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

EXPERIMENT: Watching Priority Boosts on GUI Threads

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

1. Open the System utility in Control Panel (or right-click My Computer and select Properties), click the Advanced tab, and click the Performance Options button. If you're running Windows XP or Windows Server 2003 select the Advanced tab and ensure that the Programs option is selected; if you're running Windows 2000 ensure that the Applications option is selected. This causes *PsPrioritySeparation* to get a value of 2.
2. Run Notepad from the Start menu by selecting Programs/Accessories/Notepad.
3. Run the Windows NT 4 Performance Monitor (Perfmon4.exe in the Windows 2000 resource kits). This older version of the Performance tool is needed for this experiment because it can query performance counter values at a faster frequency. (The Windows Performance tool has a maximum interval of once per second.)
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add To Chart dialog box.
5. Select the Thread object, and then select the Priority Current counter.
6. In the Instance box, scroll down the list until you see Notepad thread 0. Click it, click the Add button, and then click the Done button.
7. As in the previous experiment, select Chart from the Options menu. Change the Vertical Maximum to 16 and the Interval to 0.010, 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 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 quantums. However, the boost of 2 that is applied because it's the foreground process remains as long as Notepad remains in the foreground.
12. When you've finished, exit Performance Monitor and Notepad.

Priority Boosts for CPU Starvation

Imagine the following situation: you have a priority 7 thread that's running, preventing a priority 4 thread from ever receiving CPU time; however, a priority 11 thread is waiting for some resource that the priority 4 thread has locked. But because the priority 7 thread in the middle is eating up all the CPU time, the priority 4 thread will never run long enough to finish whatever it's doing and release the resource blocking the priority 11 thread. What does Windows do to address this situation? Once per second, the *balance set manager* (a system thread that exists primarily to perform memory management functions and is described in more detail in Chapter 7) 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. On Windows 2000 and Windows XP, the thread quantum is set to twice the process quantum. On Windows Server 2003, the quantum is set to 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.

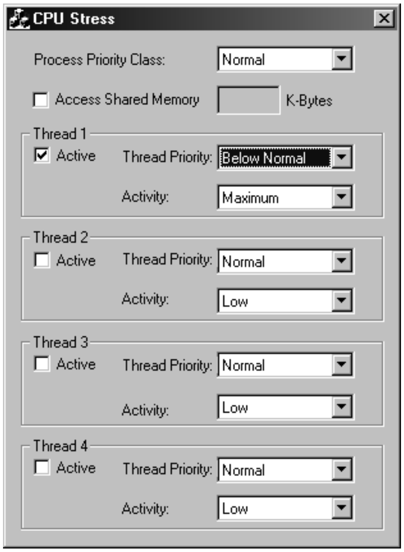
The balance set manager doesn't actually scan all ready threads every time it runs. To minimize the CPU time it uses, it scans only 16 ready threads; if there are more threads at that priority level, it remembers where it left off and picks up again on the next pass. Also, it will boost only 10 threads 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.

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 (in the resource kit and the Platform SDK), 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

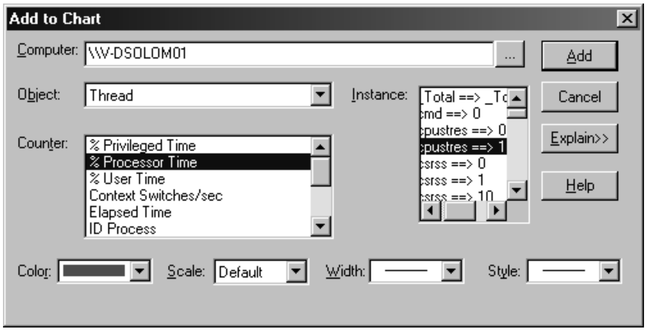
Thread 2
□ Active Thread Priority: Normal
Activity: Low

Thread 3
□ Active Thread Priority: Normal
Activity: Low

Thread 4
□ Active Thread Priority: Normal
Activity: Low

2. Run the Windows NT 4 Performance Monitor (Perfmon4.exe in the Windows 2000 resource kits). Again, you need the older version for this experiment because it can query performance counter values at a frequency faster than once per second.
3. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add To Chart dialog box.
4. Select the Thread object, and then select the % Processor Time counter.

5. In the Instance box, scroll down the list 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 to Chart

Computer: \\V-DSOLOM01

Object: Thread

Counter:
% Privileged Time
% Processor Time
% User Time
Context Switches/sec
Elapsed Time
ID Process

Instance:
_Total ==> _To
cmd ==> 0
cpustres ==> 0
cpustres ==> 1
csrss ==> 0
csrss ==> 1
csrss ==> 10

Color:
Scale: Default
Width:
Style:

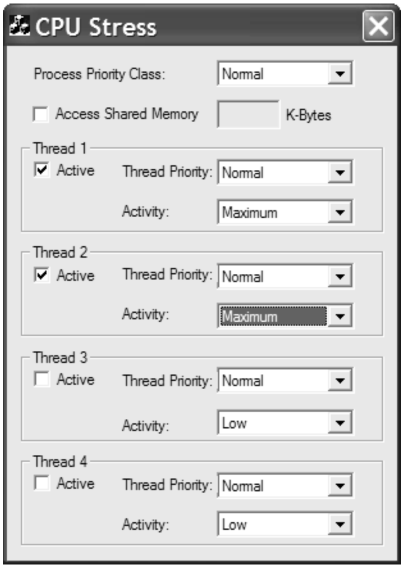
6. Click the Add button, and then click the Done button.

7. Raise the priority of Performance Monitor to real-time by running Task Manager, clicking the Processes tab, and selecting the Perfmon4.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.)

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 4 or so seconds because the thread is boosted to priority 15.

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



CPU Stress

Process Priority Class: Normal

☐ Access Shared Memory K-Bytes

Thread 1
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 2
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 3
☐ Active Thread Priority: Normal
Activity: Low

Thread 4
☐ Active Thread Priority: Normal
Activity: Low

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. Run Windows Media Player (or some other audio playback program), and begin playing some audio content.
2. Run Cpustres from the Windows 2000 resource kits, and set the activity level of thread 1 to maximum.
3. Raise the priority of thread 1 from Normal to Time Critical.
4. You should hear the music playback stop as the compute-bound thread begins consuming all available CPU time.
5. 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.
6. Stop Cpustres and Windows Media Player.

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 new types of multiprocessor systems supported by Windows (hyperthreaded and NUMA).

Multiprocessor Dispatcher Database

As explained in the "Dispatcher Database" section earlier in the chapter, the dispatcher database refers to the information maintained by the kernel to perform thread scheduling. As shown in Figure 6-16, on multiprocessor Windows 2000 and Windows XP systems, the ready queues and ready summary have the same structure as they do on uniprocessor systems. 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 DPC/dispatch level on Windows 2000 and Windows XP and to both DPC/dispatch level and Synch level on Windows Server 2003), 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.) On Windows 2000 and Windows XP, two kernel spinlocks are used to synchronize access to thread dispatching: the *dispatcher* spinlock (*KiDispatcherLock*) and the *context swap* spinlock (*KiContextSwapLock*). The former is held while changes are made to structures that might affect which thread should run. The latter is held after the decision is made but during the thread context swap operation itself.

To improve scalability including thread dispatching concurrency, Windows Server 2003 multiprocessor systems have per-processor dispatcher ready queues, as illustrated in Figure 6-24. In this way, on Windows Server 2003, each CPU can check its own ready queues for the next thread to run without having to lock the systemwide ready queues.

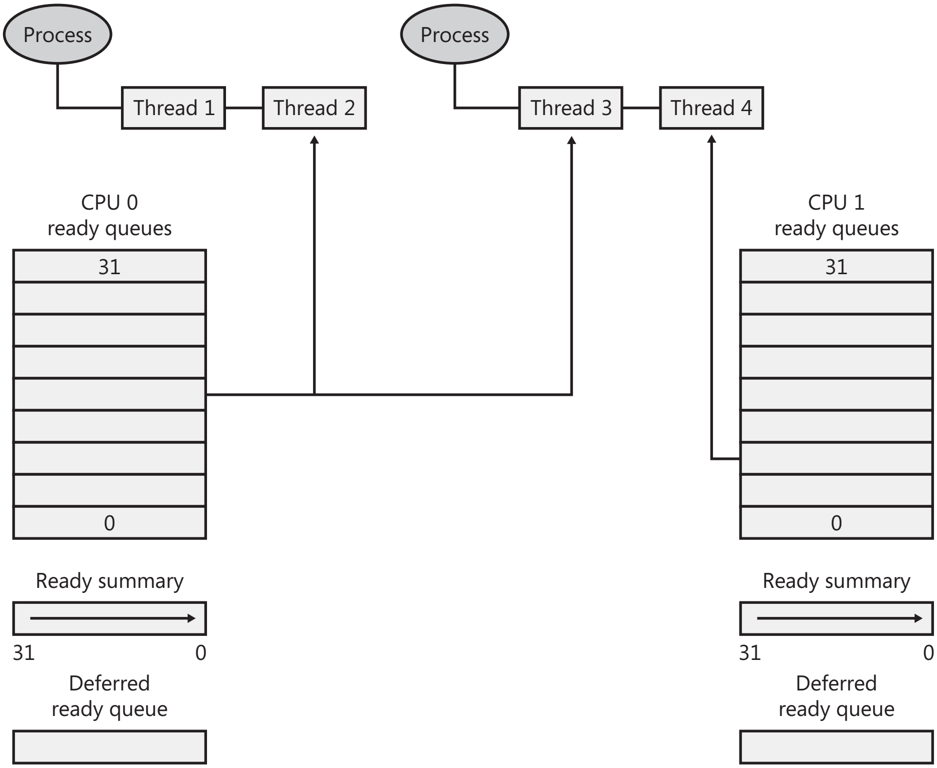


Figure 6-24 Windows Server 2003 multiprocessor dispatcher database

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.) 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 6-18 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 2000 and Windows XP, where the systemwide dispatcher spinlock had to be held.

Table 6-17 **IRQL SYNCH_LEVEL on Multiprocessor Systems**

Windows Version	IRQL
Windows 2000	2
Windows XP on x86	2
Windows Server 2003 on x86	27
Windows XP on x64	12
Windows XP on IA-64	2
Windows Server 2003 on x64 & IA-64	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-process ready queue for their priority level.

Note that the systemwide dispatcher spinlock still exists and is used on Windows Server 2003, 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.

Finally, synchronization of thread context switching has also been improved on Windows Server 20003, as it is now synchronized by using a per-thread spinlock, whereas in Windows 2000 and Windows XP context switching was synchronized by holding a systemwide context swap spinlock.

Hyperthreaded Systems

As described in the “Symmetric Multiprocessing” section in Chapter 2, Windows XP and Windows Server 2003 support hyperthreaded multiprocessor systems in two primary ways:

1. Logical processors do not count against physical processor licensing limits. For example, Windows XP Home Edition, which has a licensed processor limit of 1, will use both logical processors on a single processor hyperthreaded 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 processor 0 and 1 are on separate physical processors (as indicated by the term “Master”).

NUMA Systems

Another type of multiprocessor system supported by Windows XP and Windows Server 2003 are those with nonuniform memory access (NUMA) architectures. 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 ProcessorMask    : Uint4B
   +0x004 Color            : Uint4B
   +0x008 MmShiftedColor   : Uint4B
   +0x00c FreeCount        : [2] Uint4B
   +0x018 DeadStackList    : _SLIST_HEADER
   +0x020 PfnDereferenceSListHead : _SLIST_HEADER
   +0x028 PfnDeferredList  : Ptr32 _SINGLE_LIST_ENTRY
   +0x02c Seed             : UChar
   +0x02d Flags            : _flags
```

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 6-18. Functions that can alter thread affinity are listed in Table 6-13.

Table 6-18 NUMA-Related Functions

Function	Description
GetNumaHighestNodeNumber	Retrieves the node that currently has the highest number.
GetNumaNodeProcessorMask	Retrieves the processor mask for the specified node.
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 7).

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:

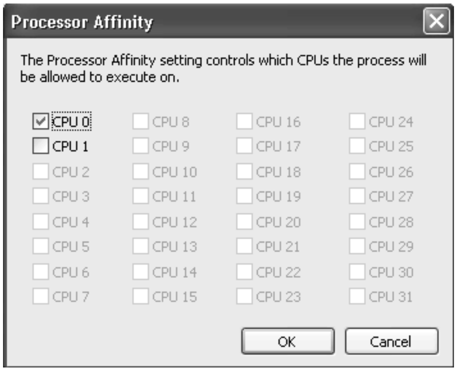
- Calling the *SetThreadAffinityMask* function to set the affinity for an individual thread
- Calling the *SetProcessAffinityMask* function to set the affinity for all the threads in a process. Task Manager and Process Explorer provide a GUI interface to this function if you right-click a process and choose Set Affinity. The Psexec tool (from *www.sysinternals.com*) 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 using, for example, the Imagecfg tool in the Windows 2000 Server Resource Kit Supplement 1 (For more information on the detailed format of Windows images, see the article "Portable Executable and Common Object File Format Specification" in the MSDN Library.)

You can also set the "uniprocessor" flag for an image (using the Imagecfg –*u* switch). 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 8	☐ CPU 16	☐ CPU 24
☐ CPU 1	☐ CPU 9	☐ CPU 17	☐ CPU 25
☐ CPU 2	☐ CPU 10	☐ CPU 18	☐ CPU 26
☐ CPU 3	☐ CPU 11	☐ CPU 19	☐ CPU 27
☐ CPU 4	☐ CPU 12	☐ CPU 20	☐ CPU 28
☐ CPU 5	☐ CPU 13	☐ CPU 21	☐ CPU 29
☐ CPU 6	☐ CPU 14	☐ CPU 22	☐ CPU 30
☐ CPU 7	☐ CPU 15	☐ CPU 23	☐ CPU 31

4. Select a subset of the available processors on the system, and press 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.

EXPERIMENT: Changing the Image Affinity

In this experiment (which requires access to a multiprocessor system), you will change the affinity mask of a program to force it to run on the first processor:

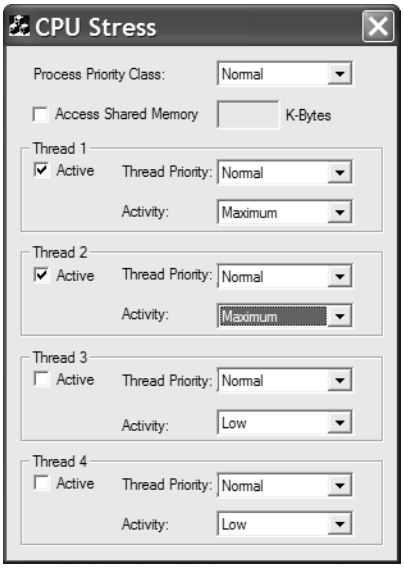
1. Make a copy of Cpustres.exe from the Windows 2000 resource kits. For example, assuming there is a c:\temp folder on your system, from the command prompt, type the following:

```
copy c:\program files\resource kit\cpustres.exe c:\temp\cpustres.exe
```

2. Set the image affinity mask to force the process's threads to run on CPU 0 by typing the following in the command prompt (assuming that the path to the resource kit tools is in your path):

```
imagecfg –a 1 c:\temp\cpustres.exe
```

3. Now run the modified Cpustres from the c:\temp folder.
4. Enable two worker threads, and set the activity level for both threads to Maximum (not Busy). The Cpustres screen should look like this:



CPU Stress

Process Priority Class: Normal

□ Access Shared Memory K-Bytes

Thread 1
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 2
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 3
□ Active Thread Priority: Normal
Activity: Low

Thread 4
□ Active Thread Priority: Normal
Activity: Low

5. Find the Cpustres process in Process Explorer or Task Manager, right click, and choose Set Affinity. The affinity settings should show the process bound to CPU 0.

6. Examine the systemwide CPU usage by clicking Show, System Information (if running Process Explorer) or by clicking the Performance tab (if running Task Manager). Assuming there are no other compute-bound processes, you should see the total percentage of CPU time consumed, approximately 1/# CPUs (for example, 50% on a dual-CPU system, 25% on a four-CPU system), because the two threads in Cpustres are forced to run on a single processor, leaving the other processor(s) idle.

7. Finally, change the affinity mask of the Cpustres process to permit it to run on all CPUs. Go back and examine the systemwide CPU usage. You should see 100% on a dual-CPU system, 50% on a four-CPU system, and so forth.

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.

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).

On Windows 2000, if none of these CPUs are idle, the first idle processor the thread can run on is selected by scanning the idle processor mask from highest to lowest CPU number.

On Windows XP and Windows Server 2003, the idle processor selection is more sophisticated. First, the idle processor set is set to the 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.

In the set of idle processors remaining, any CPUs in a sleep state are eliminated from consideration. (Again, this reduction is not performed if that would eliminate all possible processors.) Finally, the lowest numbered CPU in the remaining set is selected as the processor to run the thread on.

Regardless of the Windows version, 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. On Windows 2000, a thread's affinity mask can exclude the ideal processor. (This condition is not allowed as of Windows XP.) If that is the case, Windows 2000 selects the thread's previous processor. If that processor is not in the thread's affinity mask, the highest processor number that the thread can run on is selected.

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 in Windows Server 2003, threads are always put on their ideal processor's per-processor ready queues.

Selecting a Thread to Run on a Specific CPU (Windows 2000 and Windows XP)

In several cases (such as when a thread enters a wait state, lowers its priority, changes its affinity, or delays or yields execution), Windows must find a new thread to run on the CPU that the currently executing thread is running on. As described earlier, on a single-processor system, Windows simply picks the first thread in the highest-priority nonempty ready queue. On a multiprocessor system, however, Windows 2000 and Windows XP don't simply pick the first thread in the ready queue. Instead, they look for a thread in the highest-priority nonempty read queue that meets one of the following conditions:

- Ran last on the specified processor
- Has its ideal processor set to the specified processor
- Has been ready to run for longer than 3 clock ticks
- Has a priority greater than or equal to 24

Threads that don't have the specified processor in their hard affinity mask are skipped, obviously. If there are no threads that meet one of these conditions, Windows picks the thread at the head of the ready queue it began searching from.

Why does it matter which processor a thread was last running on? As usual, the answer is speed—giving preference to the last processor a thread executed on maximizes the chances that thread data remains in the secondary cache of the processor in question.

Selecting a Thread to Run on a Specific CPU (Windows Server 2003)

Because each processor in Windows Server 2003 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.

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 6-20 lists the Windows functions to create and manipulate job objects.

Table 6-19 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.

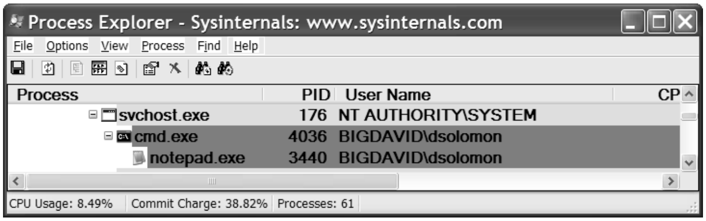
Windows 2000 Datacenter Server has a tool called the Process Control Manager that allows an administrator to define job objects, the various quotas and limits that can be specified for a job, and which processes, if run, should be added to the job. A service component monitors process activity and adds the specified processes to the jobs. Note that this tool is no longer shipped with Windows Server 2003 Datacenter Edition, but will remain on the system if a Windows 2000 Datacenter Server is upgraded to Windows Server 2003 Datacenter Edition.

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 on Windows XP and Windows Server 2003, 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:

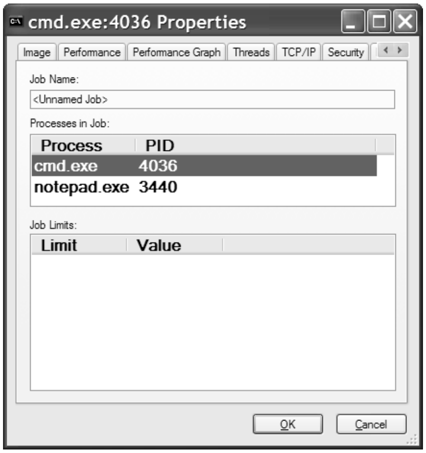


Process	PID	User Name	CP
svchost.exe	176	NT AUTHORITY\SYSTEM	
cmd.exe	4036	BIGDAVID\dsolomon	
notepad.exe	3440	BIGDAVID\dsolomon	

CPU Usage: 8.49% | Commit Charge: 38.82% | Processes: 61

4. Double-click either the Cmd.exe or Notepad.exe process to bring up the process properties. You will see a Job tab on 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	4036
notepad.exe	3440

Job Limits:

Limit	Value

6. Now run the kernel debugger on the live system (either WinDbg in local kernel debugging mode or LiveKd if you are on Windows 2000), 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 [ 0x805455c8 - 0x85797888 ]
   +0x018 ProcessListHead  : _LIST_ENTRY [ 0x8567b8dc - 0x85656324 ]
   +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 UIRestrictionsClass : 0
   +0x0b4 SecurityLimitFlags : 0
   +0x0b8 Token            : (null)
   +0x0bc Filter           : (null)
   +0x0c0 EndOfJobTimeAction : 0
   +0x0c4 CompletionPort   : 0x8619d8c0
   +0x0c8 CompletionKey    : (null)
   +0x0cc SessionId        : 0
   +0x0d0 SchedulingClass  : 5
   +0x0d8 ReadOperationCount : 0
   +0x0e0 WriteOperationCount : 0
   +0x0e8 OtherOperationCount : 0
   +0x0f0 ReadTransferCount : 0
   +0x0f8 WriteTransferCount : 0
   +0x100 OtherTransferCount : 0
   +0x108 IoInfo           : _IO_COUNTERS
   +0x138 ProcessMemoryLimit : 0
   +0x13c JobMemoryLimit   : 0
   +0x140 PeakProcessMemoryUsed : 0x256
   +0x144 PeakJobMemoryUsed : 0x1f6
   +0x148 CurrentJobMemoryUsed : 0x1f6
   +0x14c MemoryLimitsLock : _FAST_MUTEX
   +0x16c JobSetLinks      : _LIST_ENTRY [ 0x85557af4 - 0x85557af4 ]
   +0x174 MemberLevel      : 0    +0x178 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.

Many references in this chapter are to topics related to memory management. Because threads run inside processes and processes in large part define an address space, the next logical topic is how Windows performs virtual and physical memory management—the subjects of Chapter 7.