**Windows Processes and Threads**

**Process Internals - Data Structures**

Each Windows process is represented by an **executive process** (EPROCESS) structure. The Windows Kernel uses the **EPROCESS** structure to represent a process and contains all the information that the kernel needs to maintain about the process like handle table, virtual memory, security, debugging, exception, creation information, I/O transfer statistics, process timing etc, and points to a number of other related data structures.

There is an EPROCESS structure for every process running in the system including the System Process and the System Idle Process, the two processes that run in the kernel.

The Windows Kernel uses the ETHREAD structure to represent a thread and for every thread in the system there is an ETHREAD structure, including threads in the System Idle Process. The **Tcb** field contains the KTHREAD structure embedded in the ETHREAD and used to store information related to thread scheduling.

A list of all threads that belong to a process is maintained in **ThreadListHead** in which threads are queued via ETHREAD.

Every process stores the list of ETHREAD structures, representing threads running in the process.

*CreateUserProcess function opens* a valid Windows executable file and creates a section object to map it into the **new process address space**. Next it creates a Windows executive process object to run the image .

Creating the **executive process** object (which is done by the creating thread) involves the following substages:

1. Setting up the EPROCESS object
2. Creating the initial process address space
3. Initializing the kernel process control structure(**KPROCESS**)
4. Setting up the Process Environment Block (PEB)
5. Concluding the setup of the process address space (which includes initializing the working set list and virtual address space descriptors and mapping the image (executable program) into address space)

**KPROCESS**

The KPROCESS structure which is embedded inside the EPROCESS structure, and stored in the process control block (Pcb) field, is used by the lower layers of the kernel and contains scheduling related information like threads, quantum, priority and execution times.

**Overview of Windows Scheduling**

**Ref:**

[**https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=7**](https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=7)

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 setting an affinity mask in the image header.

EXPERIMENT: Viewing Ready Threads

You can view the list of ready threads with the kernel debugger *!ready* command. This command displays the thread or list of threads that are ready to run at each priority level. In the following example, generated on a 32-bit machine with a dual-core processor, five threads are ready to run at priority 8 on the first processor, and three threads at priority 10, two threads at priority 9, and six threads at priority 8 are ready to run on the second processor. Determining which of these threads get to run on their respective processor is a complex result at the end of several algorithms that the scheduler uses.

kd> !ready

**Processor 0: Ready Threads at priority 8**

THREAD 857d9030 Cid 0ec8.0e30 Teb: 7ffdd000 Win32Thread: 00000000 READY

THREAD 855c8300 Cid 0ec8.0eb0 Teb: 7ff9c000 Win32Thread: 00000000 READY

THREAD 8576c030 Cid 0ec8.0c9c Teb: 7ffa8000 Win32Thread: 00000000 READY

THREAD 85a8a7f0 Cid 0ec8.0d3c Teb: 7ff97000 Win32Thread: 00000000 READY

THREAD 87d34488 Cid 0c48.04a0 Teb: 7ffde000 Win32Thread: 00000000 READY

**Processor 1: Ready Threads at priority 10**

THREAD 857c0030 Cid 04c8.0378 Teb: 7ffdf000 Win32Thread: fef7f8c0 READY

THREAD 856cc8e8 Cid 0e84.0a70 Teb: 7ffdb000 Win32Thread: f98fb4c0 READY

THREAD 85c41c68 Cid 0e84.00ac Teb: 7ffde000 Win32Thread: ff460668 READY

**Processor 1: Ready Threads at priority 9**

THREAD 87fc86f0 Cid 0ec8.04c0 Teb: 7ffd3000 Win32Thread: 00000000 READY

THREAD 88696700 Cid 0ec8.0ce8 Teb: 7ffa0000 Win32Thread: 00000000 READY

**Processor 1: Ready Threads at priority 8**

THREAD 856e5520 Cid 0ec8.0228 Teb: 7ff98000 Win32Thread: 00000000 READY

THREAD 85609d78 Cid 0ec8.09b0 Teb: 7ffd9000 Win32Thread: 00000000 READY

THREAD 85fdeb78 Cid 0ec8.0218 Teb: 7ff72000 Win32Thread: 00000000 READY

THREAD 86086278 Cid 0ec8.0cc8 Teb: 7ff8d000 Win32Thread: 00000000 READY

THREAD 8816f7f0 Cid 0ec8.0b60 Teb: 7ffd5000 Win32Thread: 00000000 READY

THREAD 87710d78 Cid 0004.01b4 Teb: 00000000 Win32Thread: 00000000 READY

**When a thread is selected to run, it runs for an amount of time called a *quantum*. A quantum is the length of time a thread is allowed to run before another thread at the same priority level (or higher, which can occur on a multiprocessor system) is given a turn to run**. Quantum values can vary from system to system and process to process for any of three reasons: system configuration settings (long or short quantums), foreground/background status of the process, or use of the job object to alter the quantum. 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.

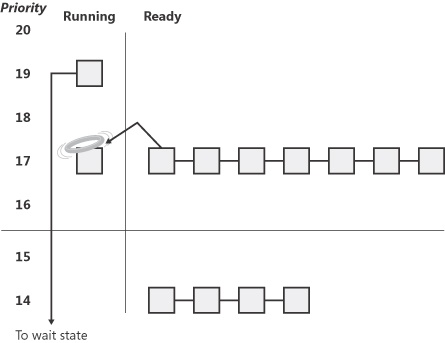
Because scheduling decisions are made strictly on a thread basis, no consideration is given to what process the thread belongs to. For example, if process *A* has 10 runnable threads, process *B* has 2 runnable threads, and all 12 threads are at the same priority, each thread would theoretically receive one-twelfth of the CPU time—Windows wouldn’t give 50 percent of the CPU to process *A* and 50 percent to process *B*.

**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.

1. **Voluntary Switch**

First a thread might voluntarily relinquish use of the processor by entering a wait state on some object by calling one of the Windows wait functions (such as *WaitForSingleObject* or *WaitForMultipleObjects*



1. **Preemption**

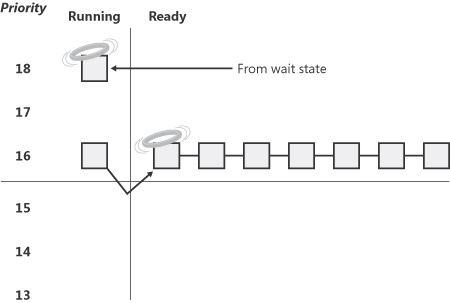
In this scheduling scenario, a lower-priority thread is preempted when a higher-priority thread becomes ready to run. This situation might occur for a couple of reasons:

* 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.

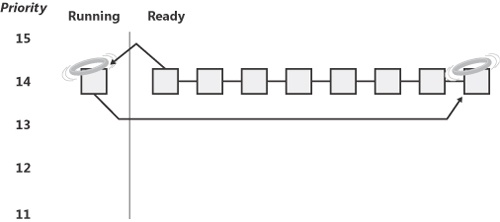
NOTE

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.



1. **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



**Threads Priority Levels**

**To understand the thread-scheduling algorithms**, you must first understand the priority levels that Windows uses. As illustrated in [Figure 5-12](javascript:popUp('/content/images/chap5_9780735625303/elementLinks/httpatomoreillycomsourcemspimages892298.jpg')), internally Windows uses 32 priority levels, ranging from 0 through 31. These values divide up as follows:

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

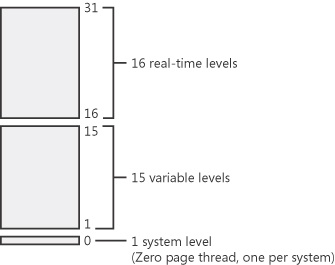


Figure 5-12. Thread priority levels

Thread priority levels are assigned from two different perspectives: those of the Windows API and those of the Windows kernel. The Windows API first organizes processes by the priority class to which they are assigned at creation (Real-time, High, Above Normal, Normal, Below Normal, and Idle) and then by the relative priority of the individual threads within those processes (Time-critical, Highest, Above-normal, Normal, Below-normal, Lowest, and Idle).

In the Windows API, each thread has a base priority that is a function of its process priority class and its relative thread priority. The mapping from Windows priority to internal Windows numeric priority is shown in [Figure 5-13](javascript:popUp('/content/images/chap5_9780735625303/elementLinks/httpatomoreillycomsourcemspimages892300.jpg')).

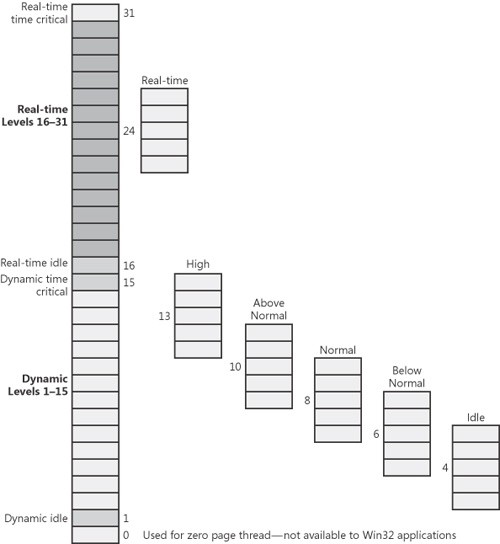


Figure 5-13. Mapping of Windows kernel priorities to the Windows API

Whereas a process has only a single base priority value, each thread has two priority values: current and base. Scheduling decisions are made based on the current priority. As explained in the following section on priority boosting, the system under certain circumstances increases the priority of threads in the dynamic range (1 through 15) for brief periods. Windows never adjusts the priority of threads in the real-time range (16 through 31), so they always have the same base and current priority.

A thread’s initial base priority is inherited from the process base priority. A process, by default, inherits its base priority from the process that created it. This behavior can be overridden on the *CreateProcess* function or by using the command-line *start* command. A process priority can also be changed after being created by using the *SetPriorityClass* function or various tools that expose that function, such as Task Manager and Process Explorer (by right-clicking on the process and choosing a new priority class). For example, you can lower the priority of a CPU-intensive process so that it does not interfere with normal system activities. Changing the priority of a process changes the thread priorities up or down, but their relative settings remain the same. It usually doesn’t make sense, however, to change individual thread priorities within a process, because unless you wrote the program or have the source code, you don’t really know what the individual threads are doing, and changing their relative importance might cause the program not to behave in the intended fashion.

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

**Windows Thread States**

Before you can comprehend the thread-scheduling algorithms, you need to understand the various execution states that a thread can be in. [Figure 5-14](javascript:popUp('/content/images/chap5_9780735625303/elementLinks/httpatomoreillycomsourcemspimages892308.jpg')) illustrates the state transitions for threads. (The numeric values shown represent the value of the thread state performance counter.) More details on what happens at each transition are included later in this section.

The thread states are as follows:

* **Ready** A thread in the ready state is waiting to execute. When looking for a thread to execute, the dispatcher considers only the pool of threads in the ready state.
* **Deferred ready** This state is used for threads that have been selected to run on a specific processor but have not yet been scheduled. This state exists so that the kernel can minimize the amount of time the systemwide lock on the scheduling database is held.
* **Standby** A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exist, the dispatcher performs a context switch to this thread. Only one thread can be in the standby state for each processor on the system. Note that a thread can be preempted out of the standby state before it ever executes (if, for example, a higher priority thread becomes runnable before the standby thread begins execution).
* **Running** Once the dispatcher performs a context switch to a thread, the thread enters the running state and executes. The thread’s execution continues until its quantum ends (and another thread at the same priority is ready to run), it is preempted by a higher priority thread, it terminates, it yields execution, or it voluntarily enters the wait state.
* **Waiting** A thread can enter the wait state in several ways: a thread can voluntarily wait for an object to synchronize its execution, the operating system can wait on the thread’s behalf (such as to resolve a paging I/O), or an environment subsystem can direct the thread to suspend itself. When the thread’s wait ends, depending on the priority, the thread either begins running immediately or is moved back to the ready state.
* **Gate Waiting** When a thread does a wait on a gate dispatcher object (see Chapter 3 for more information on gates), it enters the gate waiting state instead of the waiting state. This difference is important when breaking a thread’s wait as the result of an APC. Because gates don’t use the dispatcher lock, but a per-object lock, the kernel needs to perform some unique locking operations when breaking the wait of a thread waiting on a gate and a way to differentiate this from a normal wait.
* **Transition** A thread enters the transition state if it is ready for execution but its kernel stack is paged out of memory. Once its kernel stack is brought back into memory, the thread enters the ready state.
* **Terminated** When a thread finishes executing, it enters the terminated state. Once the thread is terminated, the executive thread block (the data structure in nonpaged pool that describes the thread) might or might not be deallocated. (The object manager sets policy regarding when to delete the object.)
* **Initialized** This state is used internally while a thread is being created.

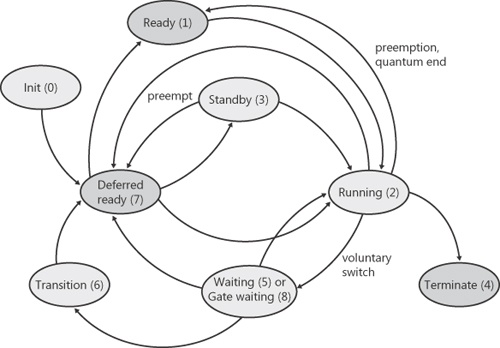


Figure 5-14. Thread states and transitions

**Context Switching**

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

* **Instruction pointer**
* **Kernel stack pointer**
* **A pointer to the address space in which the thread runs (the process’s page table directory)**

**The kernel saves this information from the old thread by pushing it onto the current (old thread’s) kernel-mode stack, updating the stack pointer, and saving the stack pointer in the old thread’s KTHREAD block**. **The kernel stack pointer is then set to the new thread’s kernel stack, and the new thread’s context is loaded. If the new thread is in a different process, it loads the address of its page table directory into a special processor register so that its address space is available.**

**Idle Thread**

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

Various Windows process viewer utilities report the idle process using different names. Task Manager and Process Explorer call it “System Idle Process,” while Tlist calls it “System Process.” **If you look at the EPROCESS structure’s** *ImageFileName* member, you’ll see the internal name for the process is “**Idle**.” Windows reports the priority of the idle thread as 0 (15 on x64 systems). In reality, however, the idle threads don’t have a priority level because they run only when there are no real threads to run—they are not scheduled and never part of any ready queues.

**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.**

**You can also set the “uniprocessor” flag for an image (at compile time). If this flag is set, the system chooses a single processor at process creation time and assigns that as the process affinity mask, starting with the first processor and then going round-robin across all the processors**. For example, on a dual-processor system, the first time you run an image marked as uniprocessor, it is assigned to CPU 0; the second time, CPU 1; the third time, CPU 0; the fourth time, CPU 1; and so on.

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.

**Windows Processes /Threads can be traced using the following tools:**

1. Windows tools
   1. Task Manager
   2. Performance Monitor
2. System Internals
   1. Process Explorer
   2. Process Monitor