

Microsoft®

Windows® Internals

6
SIXTH
EDITION

Part 1



Mark Russinovich
David A. Solomon
Alex Ionescu

Windows System Resource Manager

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

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

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

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

Thread States

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

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

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

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

TABLE 5-4 Thread States and Transitions

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

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

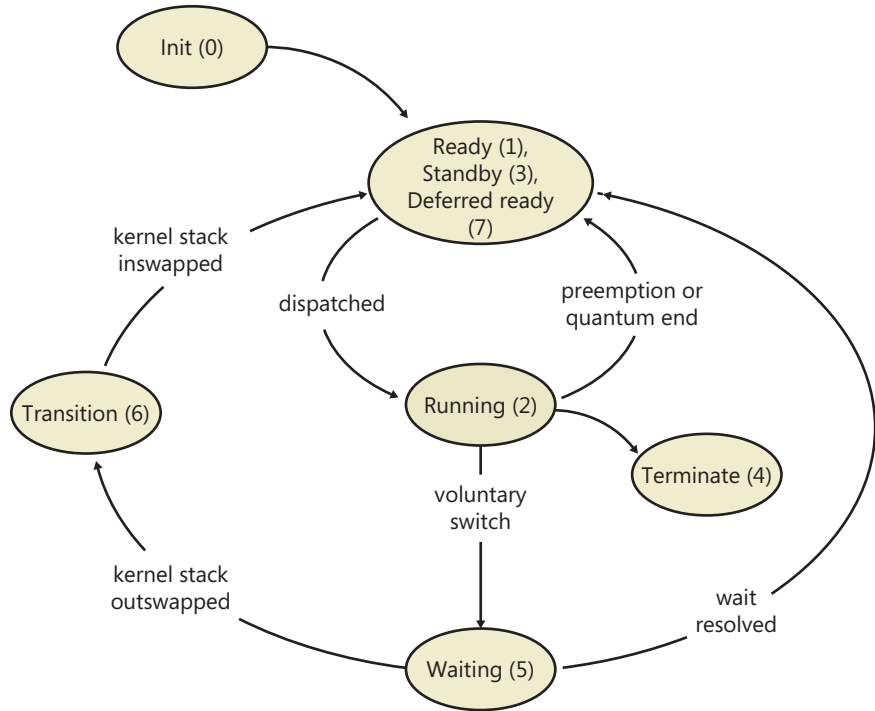


FIGURE 5-16 Simplified version of thread states and transitions