Microsoft

Windows Internals

7

SEVENTH EDITION

Part 1

System architecture, processes, threads, memory management, and more

Professional

Pavel Yosifovich
Alex Ionescu
Mark E. Russinovich
David A. Solomon

Microsoft

Windows Internals
Seventh Edition
Part 1

System architecture, processes, threads, memory management, and more

Pavel Yosifovich, Alex Ionescu,
Mark E. Russinovich, and David A. Solomon

Figure 4-8 shows the main state transitions for threads. The numeric values shown represent the internal values of each state and can be viewed with a tool such as Performance Monitor. The ready and deferred ready states are represented as one. This reflects the fact that the deferred ready state acts as a temporary placeholder for the scheduling routines. This is true for the standby state as well. These states are almost always very short-lived. Threads in these states always transition quickly to ready, running, or waiting.

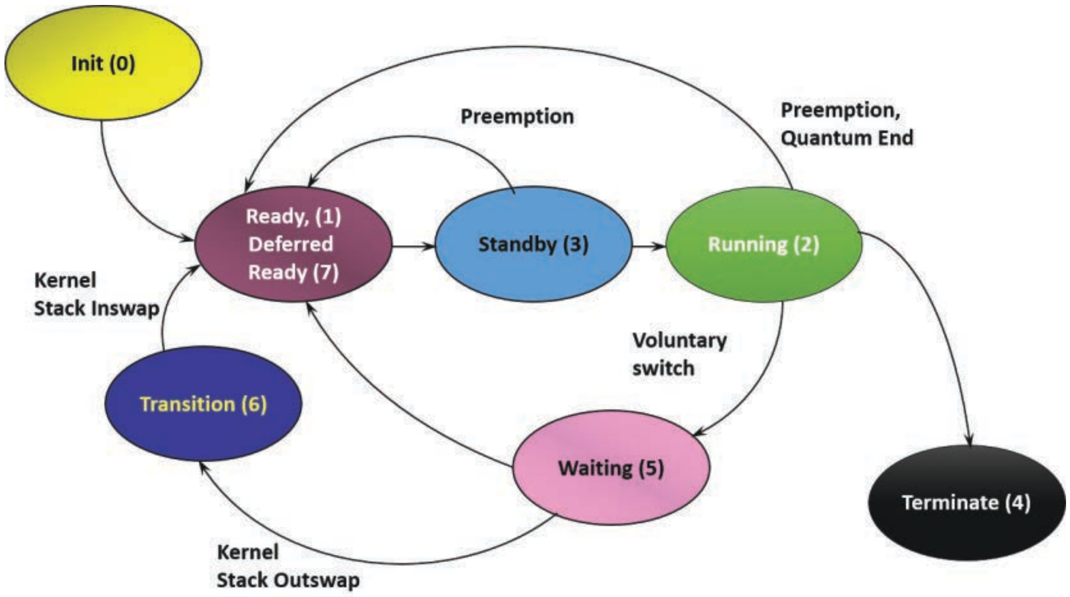
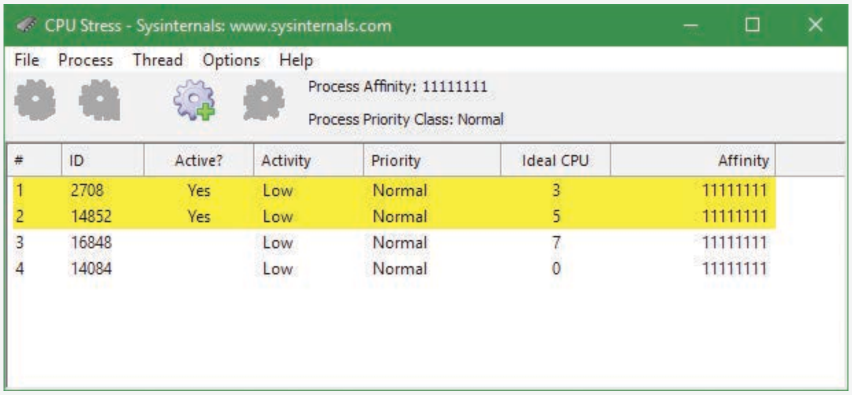


FIGURE 4-8 Thread states and transitions.

EXPERIMENT: Thread-scheduling state changes

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

1. Download the CPU Stress tool from the book's downloadable resources.
2. Run CPUSTRES.exe. Thread 1 should be active.
3. Activate thread 2 by selecting it in the list and clicking the **Activate** button or by right-clicking it and selecting **Activate** from the context menu. The tool should look something like this:

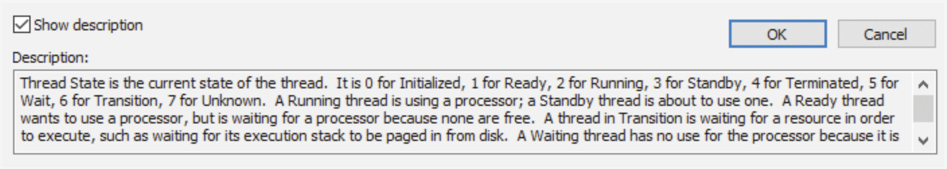

CPU Stress - Sysinternals: www.sysinternals.com
File Process Thread Options Help

Process Affinity: 11111111

Process Priority Class: Normal

#	ID	Active?	Activity	Priority	Ideal CPU	Affinity
1	2708	Yes	Low	Normal	3	11111111
2	14852	Yes	Low	Normal	5	11111111
3	16848		Low	Normal	7	11111111
4	14084		Low	Normal	0	11111111

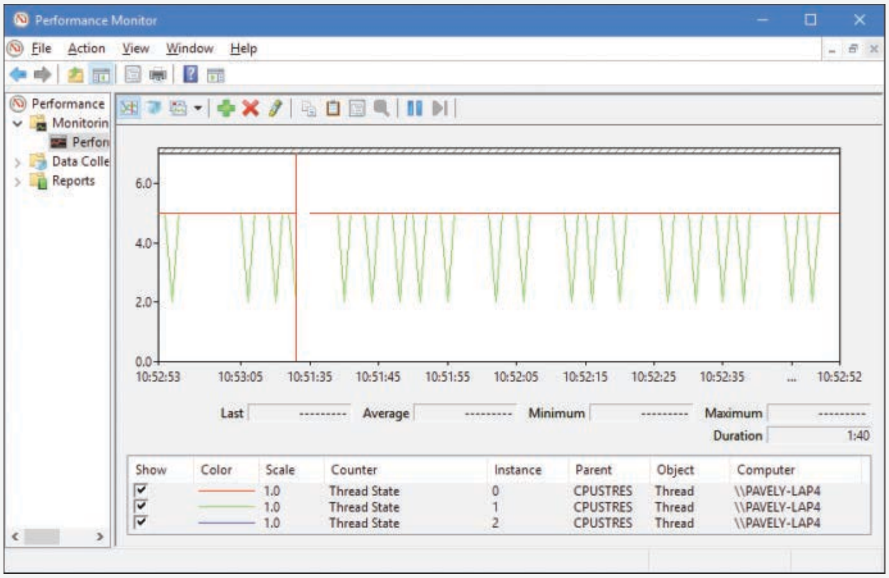
4. Click the **Start** button and type **perfmon** to start the Performance Monitor tool.
5. If necessary, select the chart view. Then remove the existing CPU counter.
6. Right-click the graph and choose **Properties**.
7. Click the **Graph** tab and change the chart vertical scale maximum to 7. (As you saw in Figure 4-8, the various states are associated with numbers 0 through 7.) Then click **OK**.
8. Click the **Add** button on the toolbar to open the Add Counters dialog box.
9. Select the **Thread** performance object and then select the **Thread State** counter.
10. Select the **Show Description** check box to see the definition of the values:



Description:

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

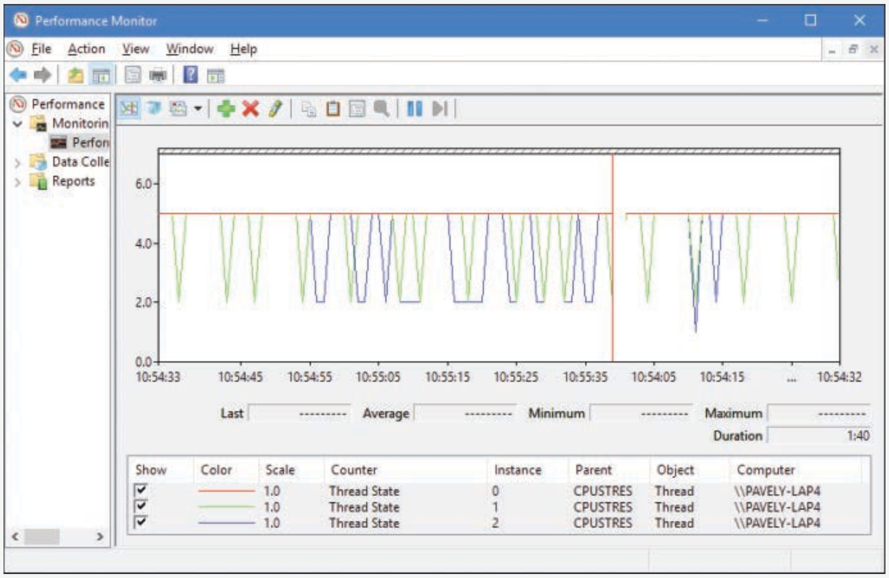
11. In the Instances box, select **<All instances>**. Then type **cpustres** and click **Search**.
12. Select the first three threads of cpustres (**cpustres/0**, **cpustres/1**, and **cpustres/2**) and click the **Add >>** button. Then click **OK**. Thread 0 should be in state 5 (waiting), because that's the GUI thread and it's waiting for user input. Threads 1 and 2 should be alternating between states 2 and 5 (running and waiting). (Thread 1 may be hiding thread 2 as they're running with the same activity level and the same priority.)



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

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1.0	Thread State	0	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟩	1.0	Thread State	1	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟦	1.0	Thread State	2	CPUSTRES	Thread	\\PAVELY-LAP4

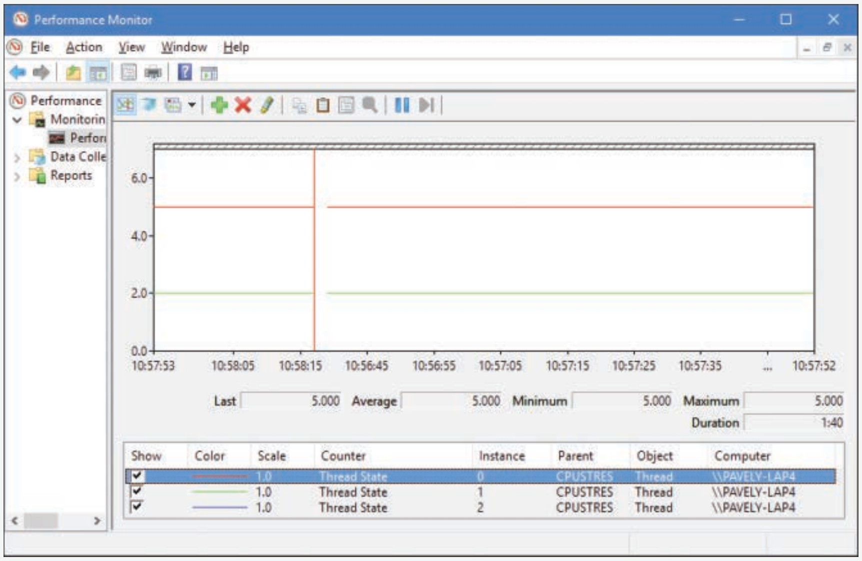
13. Go back to CPU Stress, right-click thread 2, and choose **Busy** from the activity context menu. You should see thread 2 in state 2 (running) more often than thread 1:



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

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1.0	Thread State	0	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟩	1.0	Thread State	1	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟦	1.0	Thread State	2	CPUSTRES	Thread	\\PAVELY-LAP4

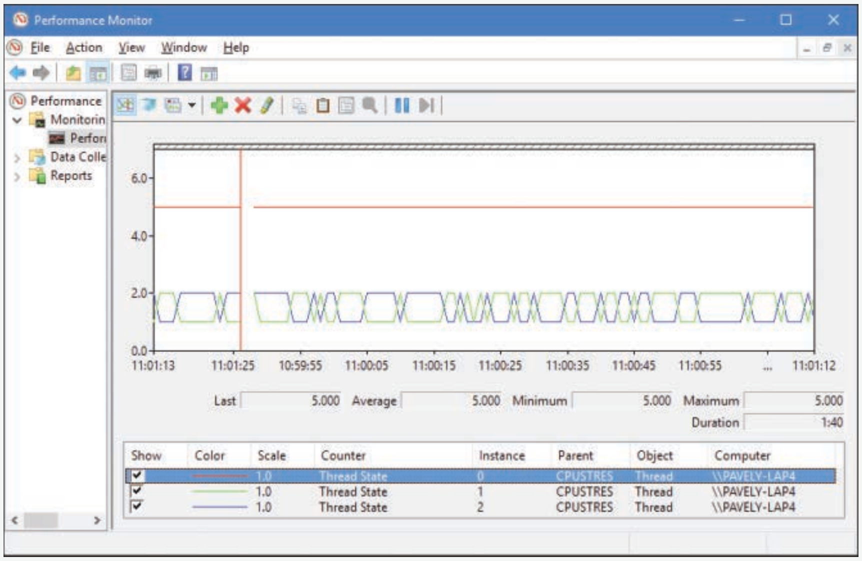
14. Right-click thread 1 and choose an activity level of **Maximum**. Then repeat this step for thread 2. Both threads now should be constantly in state 2 because they're running essentially an infinite loop:



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

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1.0	Thread State	0	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟩	1.0	Thread State	1	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟦	1.0	Thread State	2	CPUSTRES	Thread	\\PAVELY-LAP4

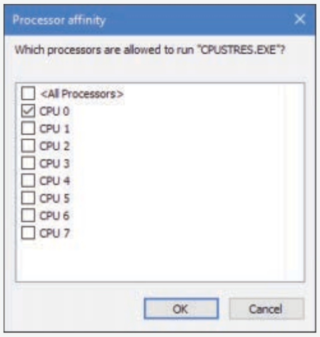
If you're trying this on a single processor system, you'll see something different. Because there is only one processor, only one thread can execute at a time, so you'll see the two threads alternating between states 1 (ready) and 2 (running):



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

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1.0	Thread State	0	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟩	1.0	Thread State	1	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟦	1.0	Thread State	2	CPUSTRES	Thread	\\PAVELY-LAP4

15. If you're on a multiprocessor system (very likely), you can get the same effect by going to Task Manager, right-clicking the CPUSTRES process, selecting **Set Affinity**, and then select just one processor—it doesn't matter which one—as shown here. (You can also do it from CPU Stress by opening the Process menu and selecting **Affinity**.)

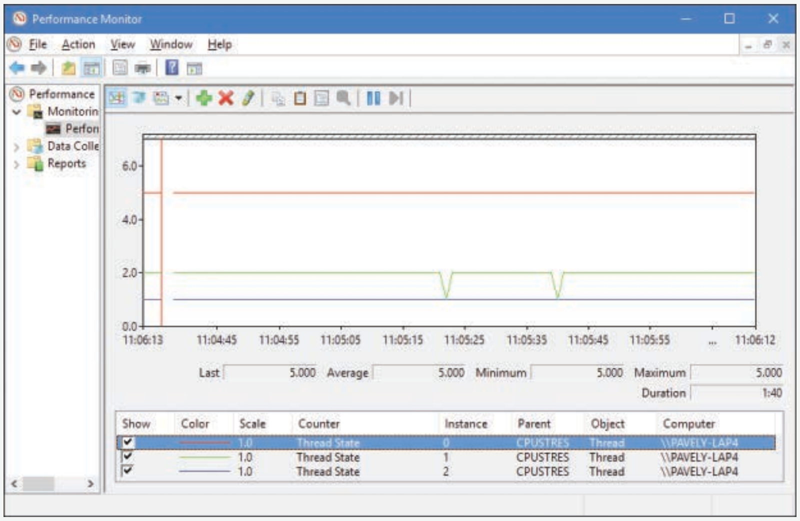


Processor affinity

Which processors are allowed to run "CPUSTRES.EXE"?

- [ ] <All Processors>
- [x] CPU 0
- [ ] CPU 1
- [ ] CPU 2
- [ ] CPU 3
- [ ] CPU 4
- [ ] CPU 5
- [ ] CPU 6
- [ ] CPU 7

16. There's one more thing you can try. With this setting in place, go back to CPU Stress, right-click thread 1, and choose a priority of **Above Normal**. You'll see that thread 1 is running continuously (state 2) and thread 2 is always in the ready state (state 1). This is because there's only one processor, so in general, the higher priority thread wins out. From time to time, however, you'll see a change in thread 1's state to ready. This is because every 4 seconds or so, the starved thread gets a boost that enables it to run for a little while. (Often, this state change is not reflected by the graph because the granularity of Performance Monitor is limited to 1 second, which is too coarse.) This is described in more detail later in this chapter in the section "Priority boosts."



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

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1.0	Thread State	0	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟩	1.0	Thread State	1	CPUSTRES	Thread	\\PAVELY-LAP4
✓	🟦	1.0	Thread State	2	CPUSTRES	Thread	\\PAVELY-LAP4

Dispatcher database

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

To improve scalability, including thread-dispatching concurrency, Windows multiprocessor systems have per-processor dispatcher ready queues and shared processor group queues, as illustrated in Figure 4-9. In this way, each CPU can check its own shared ready queue for the next thread to run without having to lock the system-wide ready queues.

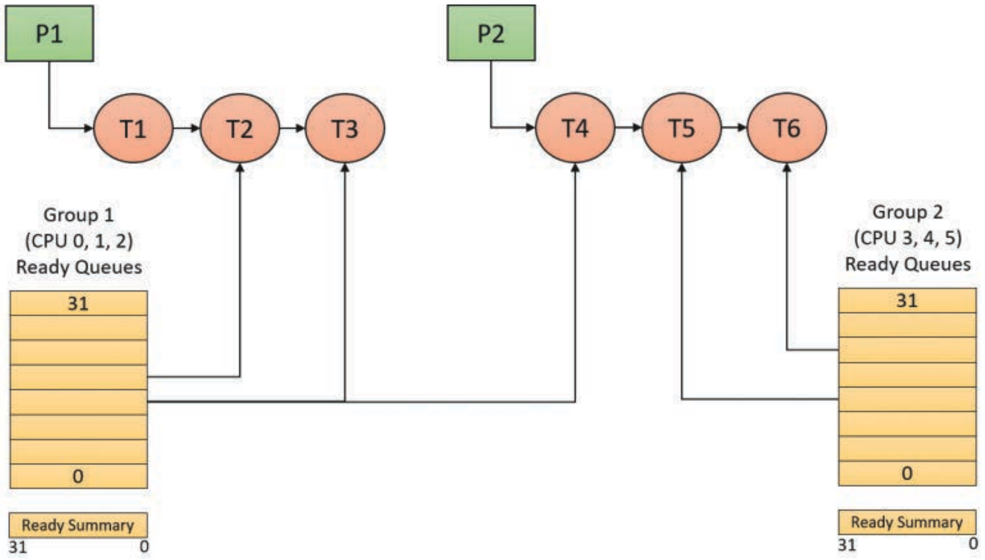


FIGURE 4-9 Windows multiprocessor dispatcher database. (This example shows six processors. *P* represents processes; *T* represents threads.)

Windows versions prior to Windows 8 and Windows Server 2012 used per-processor ready queues and a per-processor ready summary, which were stored as part of processor control block (PRCB) structure. (To see the fields in the PRCB, type **`dt nt!_kprcb`** in the kernel debugger.) Starting with Windows 8 and Windows Server 2012, a shared ready queue and ready summary are used for a group of processors. This enables the system to make better decisions about which processor to use next for that group of processors. (The per-CPU ready queues are still there and used for threads with affinity constraints.)

Note Because the shared data structure must be protected (by a spinlock), the group should not be too large. That way, contention on the queues is insignificant. In the current implementation, the maximum group size is four logical processors. If the number of logical processors is greater than four, then more than one group would be created, and the available processors spread evenly. For example, on a six-processor system, two groups of three processors each would be created.

The ready queues, ready summary (described next), and some other information is stored in a kernel structure named KSHARED_READY_QUEUE that is stored in the PRCB. Although it exists for every processor, it's used only on the first processor of each processor group, sharing it with the rest of the processors in that group.