**<https://docs.microsoft.com/en-us/previous-versions//cc750967(v=technet.10)?redirectedfrom=MSDN>**

**Chapter 13 - Detecting Processor Bottlenecks**

* 02/20/2014
* 46 minutes to read

|  |
| --- |
| **CHAPTER 13 - DETECTING PROCESSOR BOTTLENECKS** |
| *Archived content. No warranty is made as to technical accuracy. Content may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist.* |

The symptoms of a processor bottleneck aren't difficult to recognize:

* Processor: %Processor Time often exceeds 90%.
* System: Processor Queue Length is often greater than 2.
* On multiprocessor systems, System: % Total Processor Time often exceeds 50%.

But these symptoms don't always indicate a processor problem. And even when the processor is the problem, adding extra processors doesn't always solve it. In this chapter, you'll learn to use Performance Monitor to analyze such symptoms, determine the likely cause of processor bottlenecks, and implement effective solutions.

**Note** Before upgrading or adding processors, verify that the processor is the\* \*source of problem. Memory shortages, by far the most common bottleneck, often masquerade as high processor use. For more information see Chapter 12, "Detecting Memory Bottlenecks."

For more information on monitoring processor use on multiprocessor computers, see Chapter 16, "Monitoring Multiprocessor Computers."

Use the following counters to measure different aspects of processor use.

| **TABLE 2** | |
| --- | --- |
| **Object** | **Counter** |
| System | % Total Processor Time |
| System | Processor Queue Length |
| Processor | % Processor Time |
| Process | % Processor Time |
| Process | % User Time |
| Process | % Privileged Time |
| Process | Priority Base |
| Thread | Thread State |
| Thread | Priority Base |
| Thread | Priority Current |
| Thread | Context Switches/Sec |
| Thread | % User Time |
| Thread | % Privileged Time |

It is also useful to log Memory: Pages/sec, Logical Disk: % Disk Time and an activity count for your network to rule out problems in these components.

**Measuring Processor Use**

To investigate a processor bottleneck, log the System, Processor, Process, Thread, Logical Disk, and Memory counters for at least several days at an update interval of 60 seconds. Include a network counter if you suspect that network traffic might be interrupting the processor too frequently. The longer you can log, the more accurate your results will be. Processor use might be a problem only at certain times of the day, week or month, and you are likely to see these patterns if you log for a longer duration.

You can use At.exe or Microsoft Test to start and stop Performance Monitor at critical times and batch the logs for later examination.

* At.exe is included in Windows NT workstation and server. For instructions, open a command prompt window and, at the command line, type **at /?**.
* Microsoft Test is purchased separately. It is described in Chapter 11, "Performance Monitoring Tools."

You can also use CPU Stress to measure the response of your configuration to high processor use and to simulate processor bottlenecks. CPU Stress is a testing tool included on the Windows NT Resource Kit 4.0 CD in the Performance Tools group in \Perftool\Meastool\CpuStres.exe. For more information, see Rktools.hlp.

Performance Monitor includes some direct and some indirect indicators of processor use, for both single- and multiple-processor computers. This section discusses some characteristics of the measurements that you need to know to correctly interpret the values.

**The Idle Process**

Processors never rest. Once powered up, they must always be executing some thread of instructions. When not executing the thread of an active user or system process, they execute a thread of a process called *Idle*.

The Idle process has one thread per processor. It has such a low base priority that it runs only when nothing else is scheduled to run. This process does nothing but occupy the processors until a real thread is ready to use them. On a quiet machine, when you would expect processor use to be very low, the Idle process will be using most of the processor time.

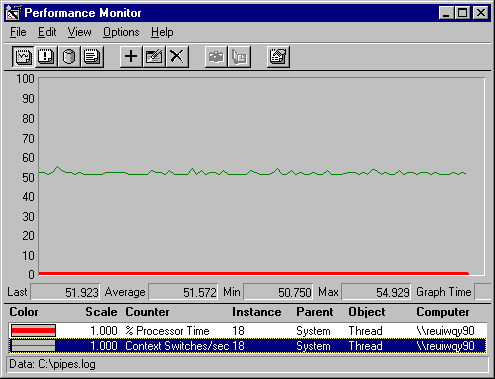
Performance Monitor and Task Manager both use the Idle thread to indicate that the processor is not busy. Processor: % Processor Time, System: % Total Processor Time, and Task Manager's CPU Usage and CPU Usage History all measure the Idle thread and display processor busy time as the difference between the total time and the time spent running the Idle thread. Performance Monitor's Process: % Processor Time for the \_Total instance even includes time processing the Idle thread.

To measure the Idle thread, use the Process: %Processor Time counter for the Idle process, or use the Processes tab on Task Manager.

**Processor Sampling**

Performance Monitor samples—rather than times—threads. Sampling uses far fewer resources, especially on Intel 486 and earlier processors which have a software timer on a separate chip. Consequently, processor time, process time, and thread time counters might underestimate or overestimate activity on your system.

The following graph demonstrates this sampling error.



In this example, the context switch rate reveals that the processor is being switched from running the System 18 thread to running other threads about 50 times each second. However, the thread's total processor time (the thick line at the bottom) appears to be 0. This contradictory data results from sampling error: the thread ran so briefly between context switches that Performance Monitor missed it.

This sampling error is most evident on processes—such as Performance Monitor—that are launched by the processor interrupt. *TotlProc*, a utility on Windows NT Resource Kit 4.0 CD, installs an extensible Performance Monitor counter designed to measure processor time on interrupt-launched applications more accurately. TotlProc is in the Performance Tools group in \PerfTool\TotlProc. For more details, see Rktools.hlp.

**Warning** TotlProc is not compatible with the processor time counters on other tools. While TotlProc is running, Performance Monitor and Task Manager processor time counters always display 100%.

**Understanding the Processor Counters**

It is important to understand the components of the primary processor activity counters, and to distinguish them from each other.

| **UNDERSTANDING THE PROCESSOR COUNTERS** | |
| --- | --- |
| **Counter** | **Description** |
| System: % Total Processor Time | For what proportion of the sample interval were all processors busy? A measure of activity on all processors. In a multiprocessor computer, this is equal to the sum of Processor: % Processor Time on all processors divided by the number of processors. On single-processor computers, it is equal to Processor: % Processor time, although the values may vary due to different sampling time. |
| System: Processor Queue Length | How many threads are ready, but have to wait for a processor? This is an instantaneous count, not an average, so it's best viewed in charts, rather than reports. Unlike disk queue counters, it counts only waiting threads, not those being serviced. The queue length counter is on the System object because there is a single queue even when there are multiple processors on the computer. |
| Processor: % Processor Time | For what proportion of the sample interval was each processor busy? This counter measures the percentage of time the thread of the Idle process is running, subtracts it from 100%, and displays the difference. This counter is equivalent to Task Manager's CPU Usage counter. |
| Processor: % User Time Processor: % Privileged Time | How often were all processors executing threads running in user mode and in privileged mode? Threads running in user mode are probably running in their own application code. Threads running in privileged mode are using operating system services. The user time and privileged time counters on the System and Processor objects do not always sum to 100%. They are measures of non-Idle time, so they sum to the total of non-idle time. For example, if the processor was running the Idle thread for 85% of the time, the sum of Processor: % User Time and Processor: % Privileged Time would be 15%. |
| Process: % Processor Time | For what proportion of the sample interval was the processor running the threads of this process? This counter sums the processor time of each thread of the process over the sample interval. |
| Process: % Processor Time: \_Total | For what proportion of the sample interval was the processor processing? This counter sums the time all threads are running on the processor, including the thread of the Idle process on each processor, which runs to occupy the processor when no other threads are scheduled. The value of Process: % Processor Time: \_Total is 100% except when the processor is interrupted. (100% processor time = Process: % Processor Time: Total + Processor: % Interrupt Time + Processor: % DPC Time) This counter differs significantly from Processor: % Processor Time, which excludes Idle. |
| Process: % User Time Process: % Privileged Time | How often are the threads of the process running in its own application code (or the code of another user-mode process)? How often are the threads of the process running in operating system code? Process: % User Time and Process: % Privileged Time sum to Process: % Processor Time. |
| Process: Priority Base | What is the base priority of the process? How likely is it that this process will be able to execute if the processor gets busy? The base priority of the process establishes a range within which the dynamic priorities of its threads vary. . The system schedules ready threads for the processor in order of their dynamic priority. |
| Thread: Thread State | What is the processor status of this thread? An instantaneous indicator of the dispatcher thread state, which represents the current status of the thread with regard to the processor. Threads in the Ready state (1) are in the processor queue. In Performance Monitor, the threads of the Performance Monitor process, Perfmon.exe always appear to be running. . Other threads that are getting processor time, are those recorded as switching from Ready (1) to Waiting (5). A table of Windows NT thread states appears in "Thread State" later in this chapter. |
| Thread: Priority Base | What is the base priority of the thread? The base priority of a thread is determined by the base priority of the process in which it runs. Except for Idle and Real-time threads, the base priority of a thread varies only +/-2 from the base priority of its process. |
| Thread: Priority Current | What is the current dynamic priority of this thread? How likely is it that the thread will get processor time? The Windows NT Microkernel schedules threads for the processor in order of their priority. . The system adjusts the dynamic priority of threads within the range of the base priority of the process to optimize the response of processes interacting with the user. |
| Thread: % Privileged Time | How often are the threads in the process running in their own application code (or the code of another user-mode process)? How often are the threads of the process running in operating system code? Process: % User Time and Process: % Privileged Time sum to Process: % Processor Time. |

**Recognizing a Processor Bottleneck**

Bottlenecks occur only when the processor is so busy that it cannot respond to requests for time. These situations are indicated, in part, by high rates of processor activity, but mainly by long, sustained queues and poor application response. If you don't have a long queue, you have a busy processor, but not a problem.

If you notice sustained high processor use and persistent, long queues:

* Rule out a *memory bottleneck*. These are much more common than processor bottlenecks, especially or workstations and small point-to-point networks. For more information, see Chapter 12, "Detecting Memory Bottlenecks."
* Identify the processes using processor time. Determine if a single process or multiple processes are active during a bottleneck.
* Examine the processor-intensive processes in detail. Determine how many threads run in the process and watch the patterns of thread activity during a bottleneck. If you develop or maintain these processes, you can write counters to monitor thread activity at a lower level.
* Consider the priority at which the process and its threads run. You may be able to eliminate a bottleneck merely by adjusting the base priority of the process or the current priorities of its threads.
* Choose a solution and test it, then log the general activity of your system, using counters like those in the Overview settings file described in Chapter 10, "About Performance Monitor."

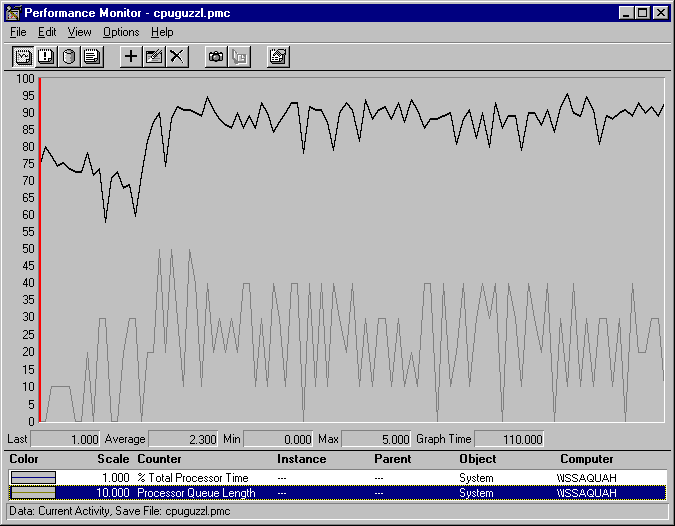
**Processor Queue Length**

The System: Processor Queue Length counter shows how many threads are contending for the processor. Threads are considered to be in the queue if they are in the Ready thread state, but not running. (Thread states are discussed in more detail later in this section.) Processor Queue Length is part of the System object, not the Processor object, because there is a single queue even when there is more than one processor.

**Tip** Start a Performance Monitor alert on System: Processor Queue Length. Set it to report an alert if the queue is over 2 and to log the alerts to the Event Viewer application event log. Review the alert panel and the logs frequently for patterns of activity that produce long queues.

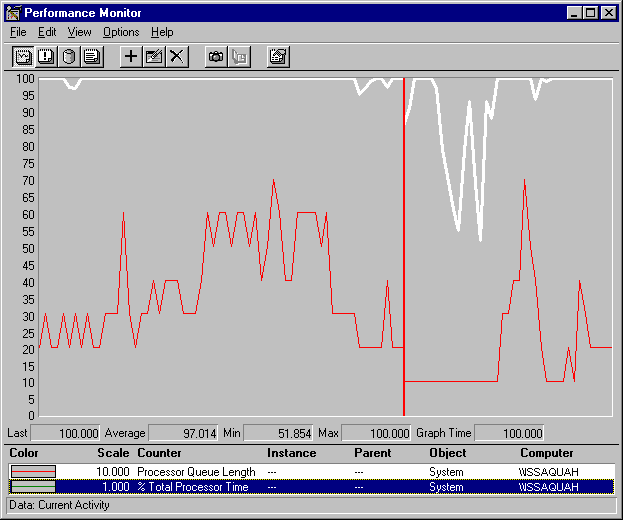
**Note** In Windows NT versions 3.5 and earlier, the Processor Queue Length counter did not work until a thread counter was added to the chart, log, or report. This was fixed in version 3.51 and is no longer necessary.

The clearest symptom of a processor bottleneck is a sustained or recurring queue of more than 2 threads. Although queues are most likely to develop when the processor is very busy, they can develop when utilization is well below 90%, and as low as 60–70%. The following graph shows a sustained processor queue with utilization ranging from 50–90%:



In this graph, the black line at the top represents System: %Total Processor Time. The gray line is System: Processor Queue Length. Queues are more likely to develop at lower processor use rates when the requests for processor time arrive in clusters or are random.

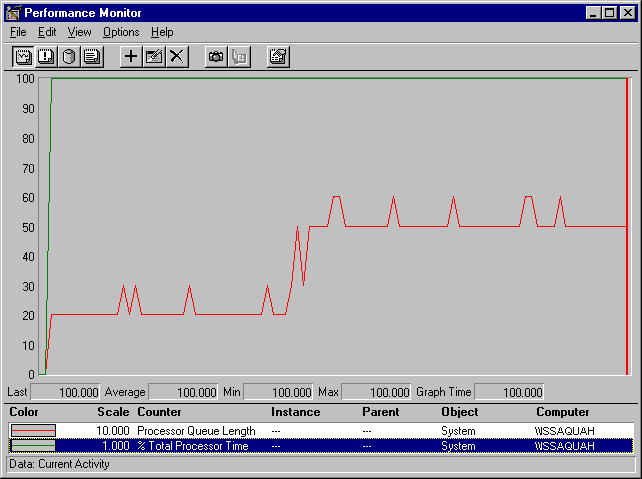
The following graph shows a sustained processor queue accompanied by processor use at or near 100%.



In this example, the queue length averages about 4 with a maximum of 7, and it never falls below 2. Note that the Processor Queue Length counter scale is multiplied by 10 to make the values easier to see. (The same effect could be achieved by reducing the vertical maximum to 10.)

If your system charts look like this, log over a longer period of time. This use pattern might be limited to a certain time of day. If so, you might be able to eliminate this bottleneck by changing the load balance between computers. However, if sustained queues appears frequently, more investigation is warranted.

The following figure uses the queue length counter to confirm a bottleneck. It shows that when a processor is already at 100% utilization, starting another process doesn't accomplish more work.



In this example, the dark line running across the top of the graph is System: % Total Processor Time. The gray line below it is System: Processor Queue Length. Midway through the sample interval, a process with three threads was started. The graph illustrates that the queue increased by three threads. Some of the threads of the added process might be in the queue, or they might be running, having displaced the threads of a lower priority process. Nonetheless, because the processor was already at maximum capacity, no more work is accomplished.

**Processes in a Bottleneck**

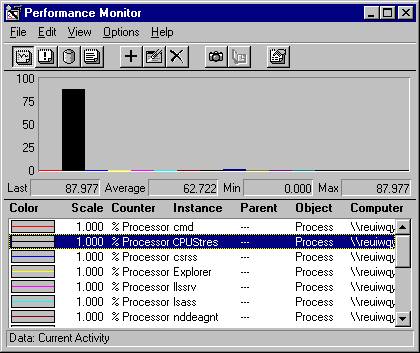
After you have recognized a processor bottleneck, the next step is to determine whether a single process is monopolizing the processor or whether the processor is consumed by running many processes.

Graphs of Processor: % Processor Time during single and multiple-process bottlenecks are nearly indistinguishable. Queue length isn't much help either, because it tells you more about what isn't running than what is. Moreover, queue length is an indicator of the numbers of threads, not the numbers of processes. The threads of a single, multithreaded process contending with each other and with other processes will produce as long a queue as the threads of several single-threaded processes.

The only clear indicator of how many processes are causing the bottleneck is a log of the processor use by each process over time. Log the Process object, then chart Process: % Processor Time for all processes except Idle and \_Total. This will reveal how many (and which) processes are very active during the bottleneck.

**Single-Process Bottlenecks**

The following figure, captured during a processor bottleneck, is a histogram of a processor bottleneck caused by a single process. This example was produced by running CPU Stress, a tool on the Windows NT Resource Kit 4.0 CD.



This histogram shows that a single process (represented by the tall, black bar) is highly active during a bottleneck; its threads are running for more than 80% of the sample interval. If this pattern persists and a long queue develops, it is reasonable to suspect that the application running in the process is causing the bottleneck.

Note that a highly active process is a problem only if a queue is developing because other processes are ready to run, but are shut out by the active process.

**Note** Histograms are useful for simplifying graphs with multiple counters. However, they display only instantaneous values, so they are recommended only when you are charting current activity and watching the graphs as they change. When you are reviewing data logged over time, line graphs are much more informative.

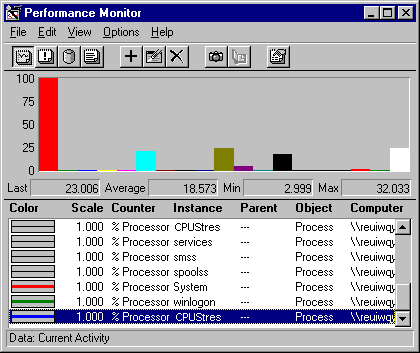
If you suspect that an application is causing a processor bottleneck:

* Stop using the application for a few days or move it to a different computer. Then, log processor use again. If the problem disappears, it is likely to have been caused by the application. If processor use continues to be a bottleneck even without the application, repeat the procedure, carefully monitoring the processes that are active when queues are longest.
* If you have traced the bottleneck to a single application, you might consider replacing it. If it is your application, you can examine its threads in detail or change its base priority. These options are described in the next sections.
* You can also use Performance Monitor to tune an inefficient application. The Win32 Software Development Kit includes tools and methods for optimizing applications, including instructions for building extensible counters to monitor the inner workings of your application.
* As a temporary solution, you can change the base priority of the active application or of an application that is being prevented from running. You can also change the relative priority of foreground to background applications. These solutions, discussed in detail in "Examining Application Priority" later in this chapter, might help you get your urgent work done while you search for a long-term solution.
* As a last resort, upgrade the processor or add additional processors. Multithreaded applications can run on multiple processors, relieving the load. However, single-threaded applications will not benefit; they must be resolved by replacing the processor with a faster one.

**Multiple-process Bottlenecks**

In a bottleneck caused by a multiple processes, no single process stands out above all others. When multiple processes are involved, several might be active, each using a smaller proportion of processor cycles. Multiprocess bottlenecks usually result when the processor cannot handle the process load. They do not usually indicate a problem with an application.

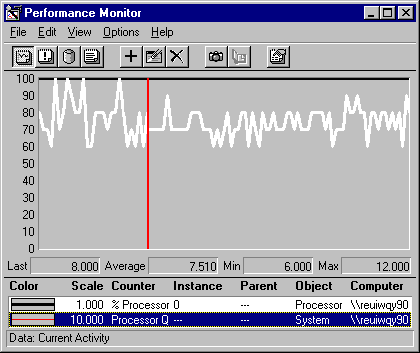
The following figure shows a histogram of processor time for many active processes.



This example was produced by using four copies of a simulation tool, CPU Stress, which consumes processor cycles at a priority and activity level you specify.

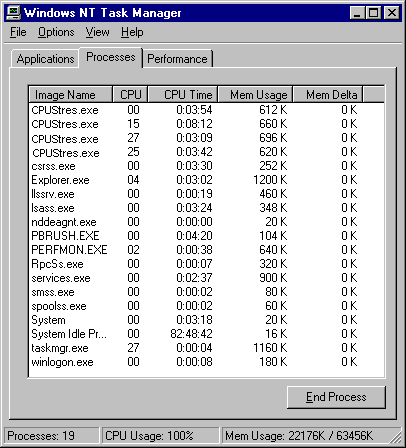
In this example, the highest bar on the far left is System: % Total Processor Time. At least four other processes (represented by bars reaching about 20% processor time) are consuming the processor while sharing it nearly equally. Although each process is only using 10–20% of the processor, the result is the same as a single process using 100% of processor time.

The following figure shows Processor: Processor Queue length during this bottleneck.



In the graph, Processor: % Processor Time (the black line running across the top of the graph) remains at 100% during the sample interval. System: Processor Queue Length (the white line) reveals a long queue. The value bar shows that the queue length varies between 6 threads and 12 threads, and averages over 7.5 threads.

The following figure shows Task Manager during the same bottleneck. It shows that four copies of CPU Stress are each using about one-fifth of the time of the single processor on the computer. (Task Manager displays current values, so you need to watch the display to see changes in processor use for each process.)



Although a faster processor might help this situation somewhat, multiple-process bottlenecks are best resolved by adding another processor. Multithreaded processes, including multithreaded Windows NT services, benefit the most from additional processors because their threads can run simultaneously on multiple processors. Even after adding another processor, it is prudent to continue testing with different priorities and processor loads to resolve this more complex situation.

**Threads in a Bottleneck**

After you've determined which process or processes are causing the bottleneck, it's time to think about threads. Threads are the components of a process that run on the processor. They are the objects that are waiting in the queue, running in user mode or operating system code, and being switched on to and off of the processor.

Understanding thread behavior is essential to understanding how processes use the processor. However, unless you are developing or maintaining an application, or have access to the person who is, there is little you can do about thread behavior.

**Warning** Performance Monitor counter values for threads are subject to error when threads are stopping and starting. Faulty values sometimes appear as large spikes in the data. For details, see "Monitoring Threads" in Chapter 10, "About Performance Monitor."

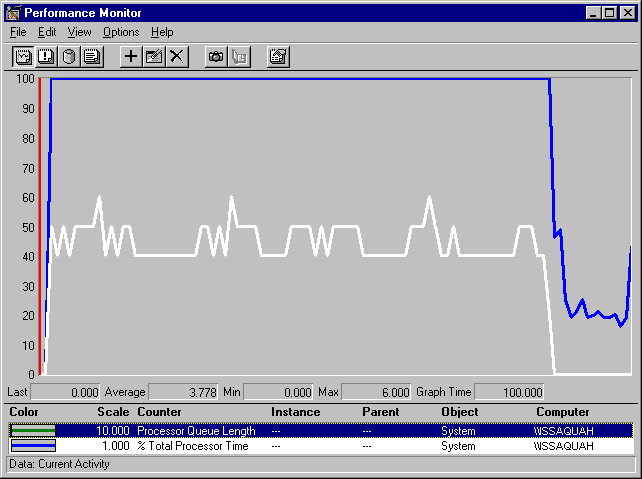
To study threads during a bottleneck, log the System, Processor, Process, and Thread objects for several days at an update interval of 60 seconds.

**Note** When logging the Thread object, you must also log the Process object. If you do not, the process names will not appear in the Instances box for the Thread object and threads will be difficult to identify.

**Single vs. Multiple Threads in a Bottleneck**

Sometimes a single thread in a process can cause a processor bottleneck, making the whole process and the processor function poorly. Bottlenecks caused by multiple threads in a single process, single threads in multiple processes, and multiple threads in multiple processes are essentially the same: Too many threads are contending for the processor at the same time. However, because these situations are resolved quite differently, it's worth distinguishing among them.

The following figure shows a graph of processor time and queue length during a bottleneck caused by a single, multithreaded process. The line running across the top is processor time. The white line is queue length.

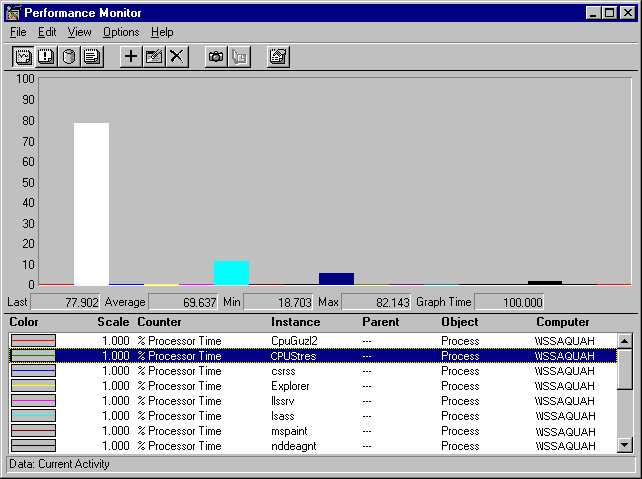


The queue is quite long, running between 4 and 5 ready threads with periodic peaks of 6 threads. The EKG-like pattern is just an artifact of the application. These large values might trick you into thinking multiple processes are at work. The clue that the queue is populated by many threads of just one process comes at the end of the graph when the process is stopped, and the queue length drops to 0.

**Uncovering Multiple Threads**

The best way to determine how many processes produced the threads in the queue is to chart the processor time used by each process.

The following figure is a histogram of Processor: % Processor Time for all processes running when the queue in the previous graph was measured:

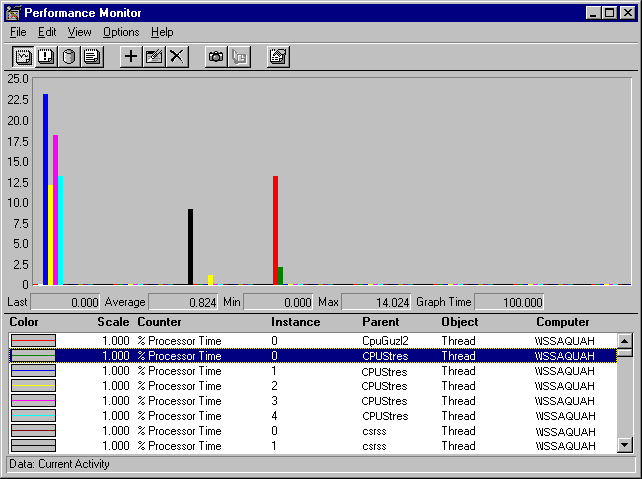


Despite the large queue, this chart makes it evident that a single process, represented by the tall, white bar, is using much more than its share of the processor—nearly 80% on average. If multiple processes were at work, no single bar would be so tall and there would be multiple bars at nearly the same height.

**Charting the Threads**

The Thread object also has a %Processor Time counter. It is most useful after you have determined that one or two processes are accounting for most of a processor's time.

The following figure is a histogram of Thread: % Processor Time for all threads running during the bottleneck.



Each bar of the histogram represents the processor time of a single thread. Threads are identified by process name and thread number and, in this graph, the threads of each process are listed in sequence. (The order in which the threads appear on the graph depends on the order in which you add them to your chart.) The thread number represents the order in which the threads started, and it can change even as the thread runs.

This graph shows that the four threads of the five threads of the CPU Stress process (at the far left) are dominating the pattern of processor use, although a few other threads are getting some processor time.

If your graphs look like these, you might consider adding another processor. A bottleneck caused by one or more multithreaded applications is a prime candidate for a multiprocessor computer. Instead, you might choose to replace the application that is consuming the processor time, or measure, tune, and rewrite it. The next sections assumes you have taken the latter path and demonstrates a more detailed investigation of thread behavior.

**Thread State**

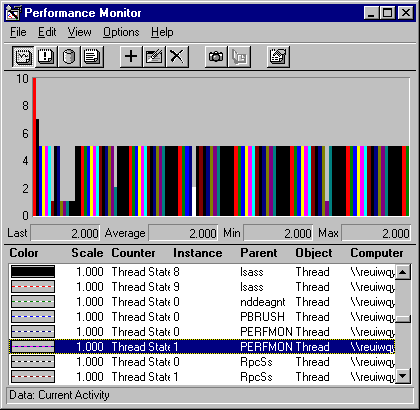
A long processor queue is a warning that a bottleneck, however brief, might be developing. The Thread: Thread State counter lets you examine which threads are in the queue and how long they remain before being serviced.

By definition, all of the threads in the processor queue are Ready, but are waiting for a processor to become available. *Ready* is a dispatcher thread state, one of eight states that signal when a thread is prepared to be dispatched to the processor.

The following table lists the thread states for threads in Windows NT.

| **THREAD STATE** | |
| --- | --- |
| **Thread state** | **Definition** |
| 0 | Initialized. The thread is recognized as an object by the microkernel. |
| 1 | Ready. The thread is prepared to run on the next free processor. |
| 2 | Running. The thread is executing. |
| 3 | Standby. The thread is assigned to a processor and about to run. Only one thread can be in Standby state at a time. |
| 4 | Terminated. The thread is finished executing. |
| 5 | Waiting. The thread is not ready for the processor. When it is ready, it will have to be rescheduled. |
| 6 | Transition. The thread is ready but waiting for resources other than the processor to become available. |
| 7 | The thread state is unknown. |

To determine which threads are contending for the processor, chart the thread states of all threads in the system. The following figure shows such a chart. The vertical maximum is reduced to 10 to show the values which range from 0 through 7.

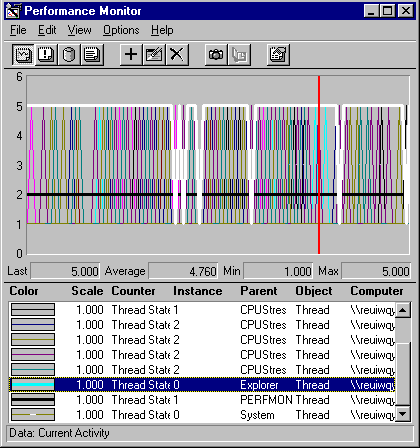


The first, tallest bar is System: % Total Processor Time, is 100%, scaled to 0.1 to fit in the chart. The next bar is System: Processor Queue Length, which is 7. The remaining bars represent the thread states of threads in active processes.

The thread that is running on the processor, that is, at Thread State 2, is PERFMON Thread 1, a thread of the Performance Monitor executable. (It is represented by the white bar in middle of the graph.) In fact, a Performance Monitor thread always appears as the running thread when it captures data; if it weren't, it could not be capturing the data. This is an inescapable artifact of the tool.

Therefore, in Thread State charts or graphs, you need to assume that the processes getting processor time are those bouncing from Ready and in the queue (1) to Waiting (5). In this example, the bars at Ready (1) are the first few on the left, representing the processor-guzzling simulation tool, a System thread, two Services threads, and an RPC subsystem thread. (As you scroll through a running Performance Monitor graph, the thread state value appears in the value bar.)

The pattern of thread state activity is better seen in a line graph. Although it is much busier, it reveals the patterns of processor use by each thread.



To create a graph like this one, chart all running threads, then delete threads that are never ready and set the vertical maximum to 6. This is a bit hard to read in still life, but the patterns for each thread become more apparent when you highlight the selected line by pressing the BACKSPACE key.

The black line that always appears to be running (at thread state 2) is PERFMON. The lines with the most activity are those of CPU Stress, the simulation tool. The white line is a thread of the Explorer process.

Although busy, this graph highlights which threads are in the queue and reveals their scheduling patterns. On logged data, you can use the Time Window to limit a thread state graph to selected data points, so you can measure the elapsed time a thread spent in each thread state. Summing the time in the ready state in each second sampled will tell you how long, on average, the threads are waiting in the queue. This information is quite useful when tuning thread behavior.

**Context Switches**

A *context switch* is when the microkernel switches the processor from one thread to another. Therefore, context switches are an indirect indicator of threads getting processor time. A careful examination of context switch data reveals the patterns of processor use for a thread and indicates how efficiently it shares the processor with other threads of the process or other processes.

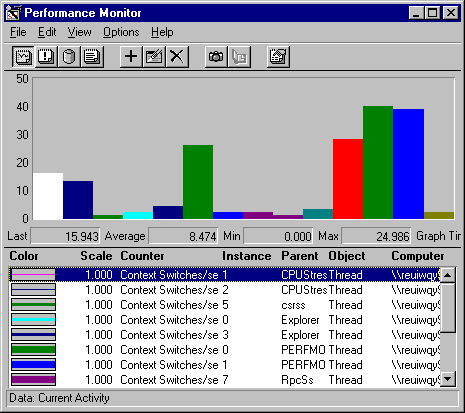
Performance Monitor has context switch counters on the System and Thread objects:

* System: Context Switches/sec counts all context switches.
* Thread: Context Switches/sec is incremented when the thread gets or loses the attention of the processor.

Both rates are an average over the last two seconds. System: Context Switches/sec and Thread: Context Switches/sec: \_Total should be identical or within the range of experimental error.

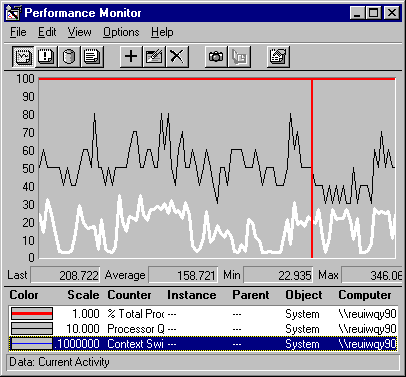
Care must be taken in interpreting the data. An application that is really monopolizing the processor actually lowers the rate of context switches because it doesn't let other process get much processor time. A high rate of context switching means that the processor is being shared, if only briefly.

The following figure is a histogram of Context Switches/sec for all running threads.



This histogram shows which threads are getting at least *some* processor time during a bottleneck. The large bars on the right side of the graph are system threads being moved onto and off of the processor during a bottleneck. The bottleneck is caused by the process represented by the first bar on the left. The process in the middle is Performance Monitor, which runs at a high priority to insure that it gets some processor time.

The following figure is a graph of System: Context Switches/sec during a transient bottleneck:



In this graph, System: % Total Processor time (the thick line running along the top of the graph) remains at 100% during the sample interval. System: Processor Queue Length (the thin line, scaled by a factor of 10), shows that the queue varies from 4 to 8, with a mean near 5. System: Context Switches (the white line), reveals an average of about 150 switches per second, a moderate rate. A much higher rate of context switches (near 500 per second) might indicate a problem with a network card or device driver.

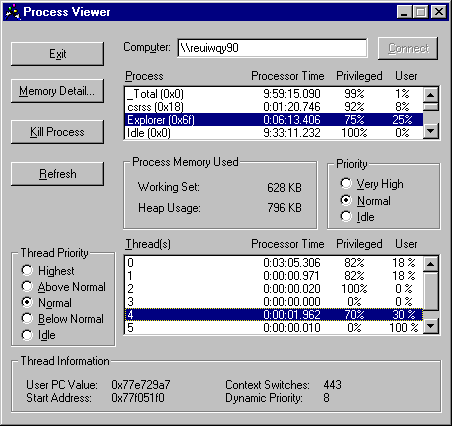
**User Mode and Privileged Mode**

Another aspect of thread behavior is whether it is running in user mode or privileged mode:

* *User mode* is the processing mode in which applications run. Threads running in user mode are running in their own application code or the code of another user mode process, such as an environment subsystem. Processes running in user mode cannot access hardware directly and must call operating system functions to switch their threads to privileged mode to use operating system services.
* *Privileged* or *kernel mode* is the processing mode that allows code to have direct access to all hardware and memory in the system. The Windows NT Executive runs in privileged mode. Application threads must be switched to privileged mode to run in operating system code. Applications call privileged-mode operating system services for essential functions such as drawing windows, receiving information about user keyboard and mouse input, and checking security.

For more information about user mode and kernel mode, see "Windows NT Workstation Architecture," earlier in this book.

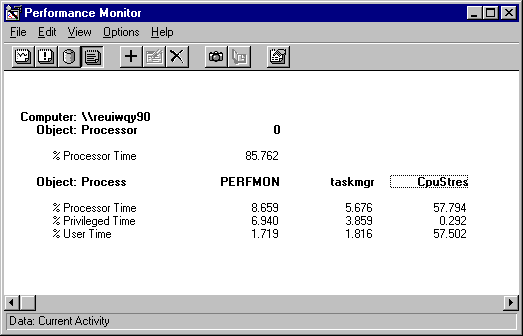
You can determine the percentage of time that threads of a process are running in user and privileged mode. Process Viewer (Pviewer.exe), a tool on the Windows NT Resource Kit 4.0 CD in the Performance Tools group, displays the proportion of user and privileged time for each running process and, separately, for each thread in the process. It can monitor local and remote computers and requires no setup. For more information about Process Viewer, see Chapter 11, "Performance Monitoring Tools," and Rktools.hlp.



Performance Monitor has % Privileged Time and % User Time counters on the System, Processor, Process, and Thread objects. These counters are described in "Understanding the Processor Counters" earlier in this chapter.

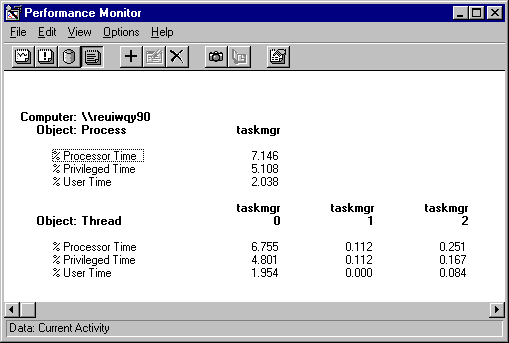
In the user time and privileged time counters, Performance Monitor displays the proportion of total processor time that the process is spending in user or privileged mode. While Process Viewer values sum to 100%, Performance Monitor values sum to their percentage of processor time.

The following figure is a Performance Monitor report on the proportion of user and privileged time for three processes.



In this example, Perfmon, the Performance Monitor process is running mainly (80%) in privileged mode, perhaps collecting data from the Performance Library which resides in the Windows NT Executive. Taskmgr, the Task Manager process is also running mainly in privileged mode (70%), though this proportion varies significantly as the process runs. In contrast, CpuStres, the process for the CPU Stress test tool, runs entirely in user mode all of the time.

The following graph shows the proportion of user and privileged time for each thread of the Task Manager process.



**Examining Priority**

Thread priority dictates the order in which threads run on the processor and, when the processor is busy, determines which threads get to run at all. The Windows NT Microkernel always schedules the highest priority ready thread to run, even if it requires interrupting a lower priority thread. This ensures that processors are always doing the highest priority task.

Examining process and thread priority is part of tuning your application and your hardware and software configuration for maximum efficiency. Windows NT adjusts thread priorities to optimize processes, but you can monitor process and thread priority, change the base priorities of processes, and change the relative priority of foreground and background applications.

**Tip** For more information about process and thread priority, including a table of all process and thread priorities, see "Scheduling and Priorities" in Chapter 5, "Windows NT Workstation Architecture."

In Windows NT, priorities are organized into a hierarchy. Each level of the priority hierarchy limits the range within which the lower levels can vary. Priorities are associated with a number from 1 to 31. The priority classes are associated with a range of numbers which sometimes overlap at the extremes.

* The *base priority class* of a process establishes a range within which the base priority of the process can vary. Processes are assigned to one of the four *base priority* classes, (Idle, Normal, High, and Real-time). The base priority class is set in application code and is not changed by the operating system.
* The *base priority of a process* varies within its base priority class. Windows NT adjust the base priority of processes to optimize the response of processes to the user. In Performance Monitor, use Process: Priority Base to monitor this value.
* The *base priority of a thread* varies within the base priority of its process. Except for Idle and Read-time threads, the base priority of a thread varies only within +/- 2 of the base priority of its process. In Performance Monitor, use Thread: Priority Base to monitor this value.
* The *dynamic priority of a thread* determines its order in processor scheduling. Windows NT continually adjusts the dynamic priorities of threads to optimize performance within the range established by the base priority. The dynamic priority of a thread can equal or exceed its base priority, but it never falls below it.

Windows NT has several strategies for optimizing application performance by adjusting process and thread priority:

* *Foreground boost*. Windows NT boosts the base priority of the process whose window is at the top of the window stack. This is known as the foreground process. It is intended to maximize the response of the process with which the user is interacting.
* *Priority inversion*. Windows NT randomly boosts the priority of lower-priority threads that would not otherwise be able to run. This is designed, in part, to prevent a lower-priority thread from monopolizing a shared resource. The lower priority thread runs long enough to free the shared resource for use by higher-priority threads. .
* *Dynamic priority boost.* When a thread has been waiting for a resource or for an I/O operation to complete, Windows NT boosts its priority as soon as the thread becomes ready. The amount of the boost depends upon what the thread was waiting for, but it is usually enough to assure that the thread runs soon thereafter, if not immediately. For example, the dynamic priority of thread waiting for disk I/O is increased by 1 when the disk I/O is complete. The dynamic priority of a thread waiting for keyboard input is increased by 5 when it is ready.

Also, if a window receives input from a timer or input device, Windows NT boosts the priorities of all threads in the process that owns the window. For example, this boost allows a thread to change the mouse pointer when the mouse moves over its window.

**Measuring and Tuning Priorities**

Windows NT and the Windows NT Resource Kit 4.0 CD include several tools for monitoring the base priority of processes and threads and the dynamic priority of threads. You can set the base priority of processes and threads in the application code. Some Windows NT tools and Resource Kit tools let you change the base priority of a process as it runs, but the change lasts only until the process stops.

**Warning** Changing priorities may destabilize the system. Increase the priority of a process may prevent other processes, including system services, from running. Decreasing the priority of a process may prevent it from running, not just allow to run less frequently.

**Tip** When you start processes from a command prompt window by using the **Start** command, you can specify a base priority for the process for that run. To see all of the start options, type **Start /?** at the command prompt.

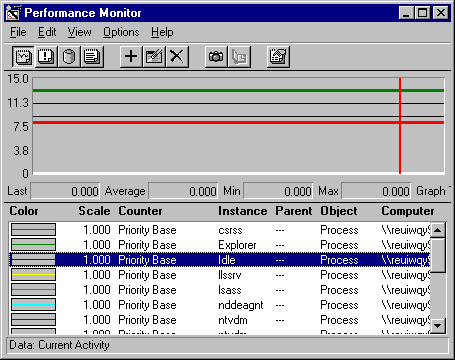
**Using Performance Monitor**

Performance Monitor lets you watch and record—but not change—the base and dynamic priorities of threads and processes. Performance Monitor has priority counters on the Process and Thread objects:

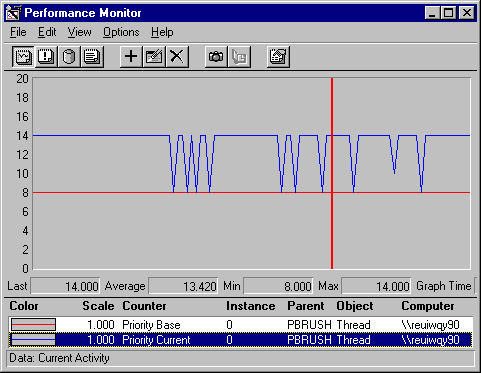
* Process: Priority Base
* Thread: Priority Base
* Thread: Priority Current

Because these counters are instantaneous and display whole number values, averages and the \_Total instance are meaningless, and are displayed as zero.

The following figure is a graph of the base priorities of several processes. It shows the relative priority of the running applications. The Idle process (the white line at the bottom of the graph) runs at a priority of Idle (0) so it never interrupts another process.



The following figure is a graph of the dynamic priority of the single thread in the Paintbrush applet, Pbrush.exe, as it changes in response to user actions. The base priority of the thread (the gray line) is 8 (foreground Normal). During this period of foreground use, the dynamic priority of the thread (the black line) is 14, but drops to 8 when other processes need to run.



**Using Task Manager**

Task Manager displays and let you change the base priority of a process, but it does not monitor threads. Base priorities changed with Task Manager are effective only as long as the process runs. For more information, see "Task Manger" in Chapter 3, "Performance Monitoring Tools."

**To display the base priorities of processes in Task Manager**

1. Open Task Manager (CTRL+SHIFT+ ESC), and click the **Processes** tab.
2. From the **View** menu, click **Select Columns**.
3. Select **Base Priority**.

**To change the base priority of a process**

1. Click the Processes tab.
2. Click a process name with the right mouse button. A menu appears.
3. Click **Set Priority**, then click a new base priority to change it.

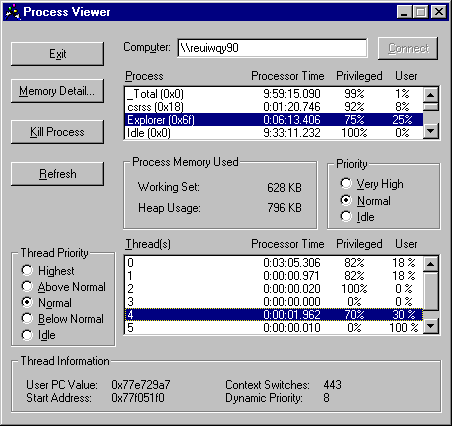
The change is effective at the next Task Manager update; you need not restart the process.

**Using Process Viewer**

Process Viewer (Pviewer.exe), a tool on the Windows NT Resource Kit 4.0 CD in the Performance Tools group (\PerfTool\MeasTool), lets you monitor process and thread priority and change the base priority class of a process. For more information, see "Process Viewer" in Chapter 2, "Performance Monitoring Tools" and Rktoools.hlp.

**Note** Process Explode (Pview.exe), also on the Windows NT Resource Kit 4.0 CD, is a superset of the functions of Process Viewer. Although the interface is different, both utilities get their information from the same source and you can change the base priority class by using either tool. For more information, see "Process Explode" in Chapter 11, "Performance Monitoring Tools."

To display the base priority of a process, open Process Viewer and select the computer you want to monitor.



**To display or change the base priority of a process**

1. In the Process box, click the line representing the process. All data in the dialog box changes to show the values for that process. The base priority of the process appears in the Priority box (under the Process box.).
2. To change the priority of the process for this run, click a button in the Priority box. The three priority classes listed—Idle, Normal, and Very High—are associated with the priority classes Idle, Normal, and High, respectively. You cannot change a process to the Real-time priority class.

**To display the dynamic priority of a thread**

1. In the Process box, click the line representing the process.
2. From the Thread box (near the bottom), select the line representing the thread. All thread data changes to show values for that thread. The Thread Priority box (to the left of the thread box) displays the thread's dynamic priority in radio buttons, but the buttons do not work: You cannot change the dynamic priority of a thread by using Process Viewer.

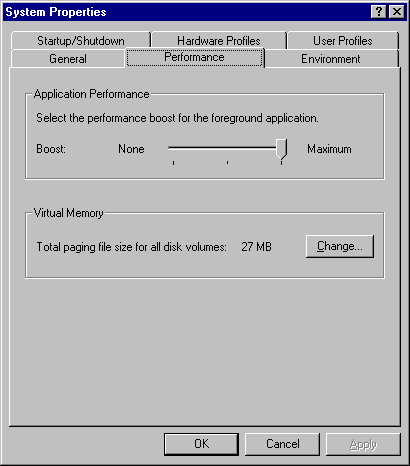
**Tuning Foreground and Background Priorities**

When a user interacts with an application, the application moves to the *foreground* and the operating system *boosts* (increases) the base priority of the process to improve its response to the user. The application returns to *background* priority—and loses the boost—when the user interacts with a different process.

You can change the amount of boost given to foreground applications and, in so doing, change the relative priority of foreground to background applications. Reducing the boost improve the response of background process when they are not getting enough processor time.

**To change the priority of foreground application**

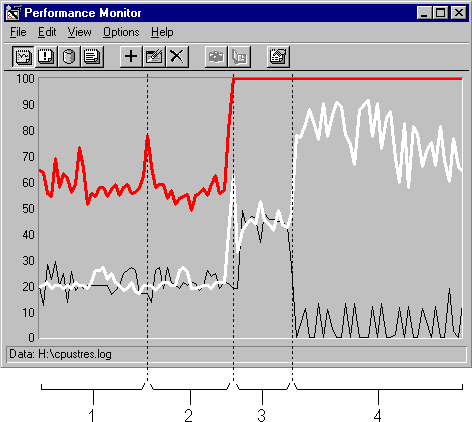
1. Double-click the Control Panel System applet and select the Performance tab. The Application Performance box shows the current setting for the boost.
2. Drag the arrow to set the boost. The default is Maximum (a priority boost of 2) but you can change it to 1 or to None (0), the setting at which foreground processes have no advantage over background processes. For example, most applications run at Normal priority (7) and are boosted to Foreground Normal priority (9) when the user moves the application window to the top of the window stack.



**Priority Bottlenecks**

When a processor has excess capacity, process and thread priority do not affect performance significantly. All threads run when ready, regardless of their priority. However, when the processor is busy, lower priority applications and services might not get the processor time they need.

The following graph shows threads of different priorities contending for processor time. It demonstrates the changing distribution of processor time among processes of different priorities as demand for processor time increases. (This test was conducted by using CPU Stress, a tool on the Windows NT Resource Kit 4.0 CD that lets you set the priorities and activity levels of a process and its threads.)



This graph shows two threads of the same process running on a single-processor computer. Lines have been added to show each of the four parts of the test. The thick, gray line is System: % Total Processor Time. The white line represents processor time for Thread 1; the white line represents processor time for Thread 2.

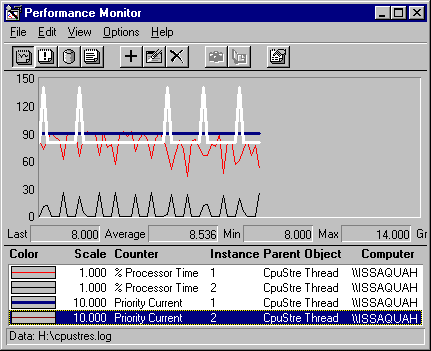
The test conditions and results are represented in the following table:

| **TABLE 5** | | | | |
| --- | --- | --- | --- | --- |
| **Object** | **Part 1** | **Part 2** | **Part 3** | **Part 4** |
| **Processor:** % Processor Time | 58.8 | 54.1 | 100 | 100 |
| **Thread 1** |  |  |  |  |
| Priority | Normal (8) | Above normal (9) | Normal (8) | Above normal (9) |
| Thread: %Processor Time | 20.6 | 21.5 | 44.2 | 76.5 |
| **Thread 2** |  |  |  |  |
| Priority | Normal (8) | Normal (8) | Normal (8) | Normal (8.5) Varies from 8 - 14 |
| Thread: %Processor Time | 20.9 | 20.8 | 45.0 | 5.6 |

This test demonstrates that when the processor has extra capacity, increasing the priority of one thread has little effect on the processor time allotted to each of the competing threads. (The small variation is not statistically significant.) However, when the processor is at its busiest, increasing the priority one of the threads, even by one priority level, causes the higher priority thread to get the vast majority of processor time (an average of 76.5%).

In fact, in Part 4, when all processor time is consumed, Thread 2 might not have been scheduled at all were it not for Windows NT's priority inversion strategy. *Priority inversion* is when the Microkernel randomly boosts the priorities of ready threads running in Idle, Normal and High priority processes so that they can execute. (It does not boost the threads of Real-Time processes.) Windows NT uses priority inversion to give processor time to lower priority ready threads which wouldn't otherwise be able to run.

The effect of priority inversion is shown in the following graph. This graph was created by using the Time Window to limit the data to Part 4 of the test. The current priority values were scaled by 10 to make them visible on the graph and the vertical maximum of the graph was increased to 150.



This graph compares priority with processor time during Part 4 of the test. The current priority of Thread 1 (the thick, black line) remains at 9 throughout the sample interval, and its processor time averages 76.5%. Although the priority of Thread 2 (the white line) was set to 8 by CPU Stress, it is repeatedly boosted to 14 in order to enable it to be scheduled. Despite the boost, it ran for an average of only 5.6% of the sample interval.

**Eliminating a Processor Bottleneck**

If you determine that you do have a processor bottleneck, consider some of these proposed solutions:

* Rule out a memory bottleneck that might be consuming the processor. Memory bottlenecks are far more common and severely degrade processor performance. For more information, see Chapter 12, "Detecting Memory Bottlenecks."
* Replace the application or try to fix it by using the performance optimizing tools in the Win32 Software Development Kit. Screen savers are notorious for monopolizing the processor, so monitor your screen saver executable.
* Upgrade your network or disk adapter cards. 8-bit cards use more processor time than 16-bit or 32-bit cards. The number of bits is the amount of data moved to memory from the adapter on each transfer, so 32-bit cards are the most efficient.

Look for cards that use bus-mastering direct memory access (DMA), not programmed I/O to move their data. Programmed I/O relies on processor instructions. In bus-mastering DMA, the disk controller managers the I/O bus and uses the DMA controller to manage DMA operation. This frees the processor for other uses.

* Increase the processor clock speed. Clock doubler and tripler processors multiply the processor clock speed while leaving the rest of the memory and I/O bus speeds alone. This will improve application response, but typically by less than the multiplier because applications do more than just use the processor.
* Increase the size of your secondary memory cache, especially if you've just added memory. When you add memory, the secondary cache must map the larger memory space, usually resulting in lower cache hit rates and more processor work retrieving data from the disk or network. Processor-bound programs are especially vulnerable because they are scattered more widely throughout memory after new memory is added.

The amount of secondary cache a system supports depends upon the design of the motherboard. Many motherboards support several secondary cache configurations (from 64K–512K or 256K–1MB). Increasing cache size usually requires removing the existing static ram (SRAM) chips, replacing them with new SRAM chips, and changing some jumpers. This is helpful when you have a working set that is larger than your current secondary cache.

* Upgrade to a faster processor or add another processor. The general rule is that a faster processor effectively improves performance when a single-threaded process is consuming the processor. Additional processors are recommended when you are running multiple processes or multithreaded processes.

**Addendum: Architectural Changes and Processor Use**

The Windows NT 4.0 Workstation and Server architecture has changed substantially from previous versions. To those monitoring performance, the most obvious change is in the behavior of Csrss.exe. This section describes how the architectural changes are likely to influence the performance of your Windows NT workstation.

For a complete discussion of the architectural changes, and the old and new components of Csrss.exe, see Chapter 5, "Windows NT Workstation Architecture."

In prior versions of Windows NT, Csrss.exe (the process for the Client Server Runtime Subsystem) included all of the graphic display and messaging functions of the Win32 environment subsystem, including the Console, Window Manager (User), the Graphics Device Interface (GDI), the user-mode graphics device drivers, and miscellaneous environment functions to support 32-bit Windows applications. With Windows NT 4.0, the User, GDI, and graphics device drivers have been moved into the Windows NT Executive. Csrss.exe now represents just the Console and miscellaneous functions that remain in the Win32 environment subsystem.

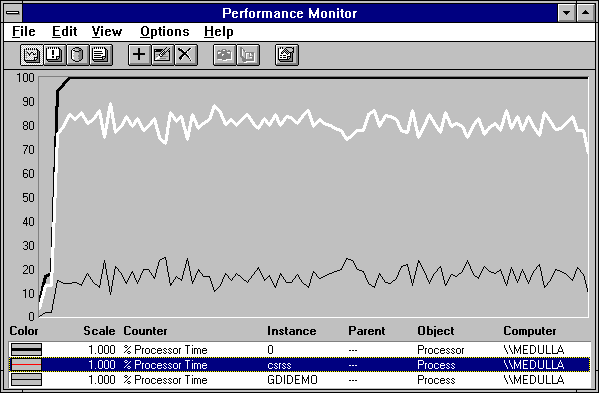
The Win32 environment subsystem runs in a special protected process in user-mode, the processor mode used by application. In Windows NT 3.51 and earlier, calls to User and GDI required the use of special fast LPCs, 64-bit shared memory windows, and multiple user and kernel mode thread transitions to communicate with the process.

Moving User, GDI, and the graphics device drivers into the Windows NT Executive, and having them run in privileged mode, eliminates these complex transitions and improves performance and memory use. Windows NT applications must still call the User32 and GDI32 functions, but now these are calls to the Windows NT Executive which requires just one kernel-mode thread transition.

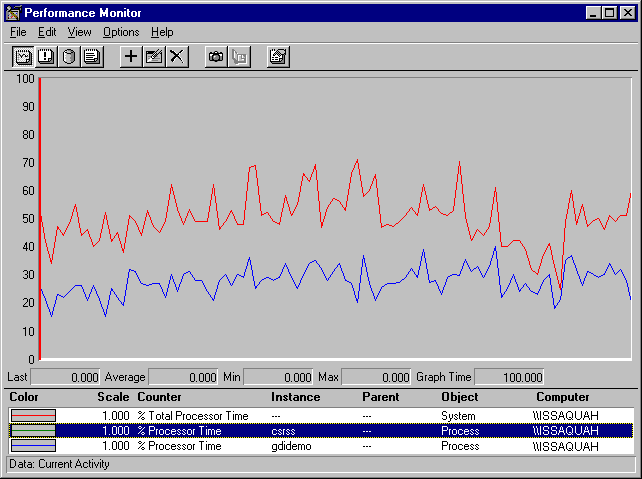
**Measuring the Change**

These architectural changes are manifest in several different ways but are most evident when you monitor the behavior of graphics-intensive applications.

The following figure is a graph of a Windows NT 3.0 running GDIDemo, a graphics demonstration tool in the Win32 Software Development Kit. All features of the tool were used simultaneously to generate the most graphics activity.



In this example, the processor is 100% busy and is spending most of its time running Csrss.exe. The following figure shows the same program running on Windows NT 4.0.



There are significant differences between the performance evidenced in the graphs.

*Processor Time:*

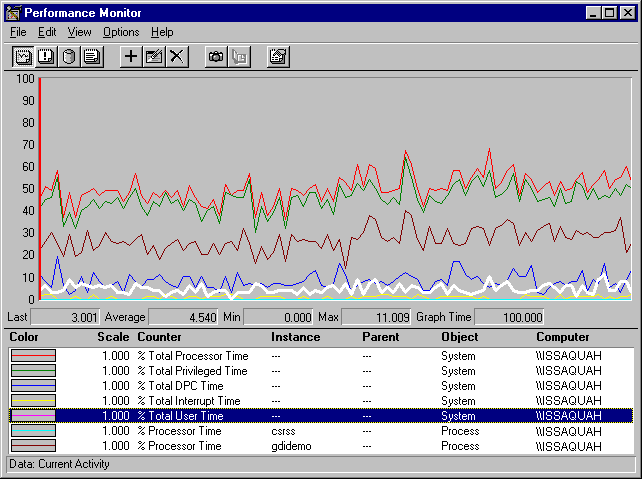
* The Windows NT 3.51 graph shows that the GDIDemo test used 100% of processor time. (System: % Total Processor Time is the thick, black line is running along the top of the graph).
* The Windows NT 4.0 graph shows that the GDIDemo test used an average of 50% of processor time. . (System: % Total Processor Time is the gray line)

Some of this might be attributable to the new architecture, but it probably results from improvement in processor technology. The Windows NT 3.51 graph was captured on a computer with a 386 processor; the Windows NT 4.0 was captured on a computer with a 486 processor.

*Process Time:*

* In the Windows NT 3.51 graph, Process: % Processor Time for GDIDemo (the thin, black line) runs between 10 and 20% while Process: % Processor Time for CSRSS (the white line) averages between 75% and 85%. This indicates that in order to run GDIDemo, the processor was spending 10–20% of its time running in GDIDemo code and the remainder running in CSRSS code on behalf of GDIDemo.
* In the Windows NT 4.0 graph, Process: % Processor Time for GDIDemo (the thin, black line) runs between 25 and 35% while Process: % Processor Time for CSRSS (the white line) runs at nearly 0%. The processor may have been running some system code on behalf of GDIDemo, but it was not in CSRSS.

The following graph of the Windows NT 4.0 test is the same as the previous one, but includes counters for privileged and user time and interrupts.



From top to bottom, the lines represent:

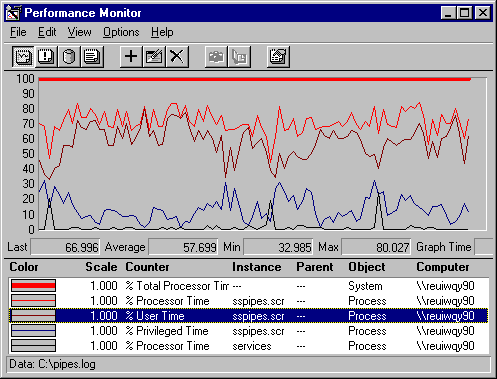
* System: % Total Processor Time
* System: % Total Privileged Time
* Process: % Processor Time: GDIDemo
* System: % Total DPC Time
* System: % Total User Time
* System: % Total Interrupt Time
* Process: % Processor Time: CSRSS

This data demonstrates that in Windows NT 4.0, GDIDemo graphics processing time is spent running in privileged mode code in the Windows NT Executive. In previous versions of Windows NT, graphics processing was part of CSRSS and ran mainly in user mode.

**Measuring 3D Pipes**

The 3D Pipes screensaver that comes with Windows NT Server and Workstation is entertaining, but it consumes substantial processor time. If you leave your computer while it's performing background work, and 3D Pipes is activated, your work must compete with the screensaver for processor time. You can measure the processor time used by the 3D Pipes process, Sspipes.scr.

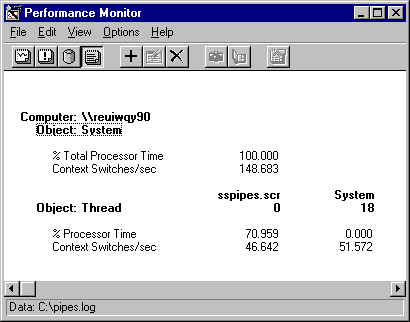
The following figure is a graph of data logged on an Intel 486 processor while 3D Pipes is running. The Time Window was adjusted to 96 data points, so all data points are shown.



In this graph, the lines, from top to bottom, follow their order in the legend. System: % Total Processor Time (the thick line at the top) is 100% throughout the sample interval. The screensaver process, Sspipes.scr, is using more than 70% of that time. Although 3D Pipes is a graphics program, it runs almost entirely in user mode (82.4% = 57.7% user mode /70% Process: % Processor time) because it is using OpenGL application services instead of operating system services.

Context switching, the rate at which the processor is switched from one thread to another, is an indicator of which threads are getting processor time. In previous version of Windows NT, it was an indirect indicator of communication between the graphics subsystem and the Win32 subsystem. Each time an application needed a graphics service, its thread was switched back and forth several times between user mode and kernel mode to access the protected Win32 subsystem. Each thread transition generated a context switch. In the new architecture, each graphics call requires a single thread transition from user to kernel mode, and back. This design generates far fewer context switches.

The following report demonstrates the effect.



Running graphics-intensive programs like GDIDemo in Windows NT 3.0 generated a sustained average of 1500 context switches per second, with more than 650 context switches per second each attributable to GDIDemo and CSRSS. Running 3D Pipes in Windows NT 4.0, System: Context Switches/sec totaled 148, just 48 more than idle. Although processor use is very high for the screensaver, processor time is consumed by elements other than context switching.