**Performance counters CategoryName**

Here a short description of some performance counters.

**1 CategoryName: Processor**

**PerformanceCounter(“Processor“, “% Processor Time“, “\_Total”);**

The **Processor\% Processor Time** counter determines the percentage of time the processor is busy by measuring the percentage of time the thread of the Idle process is running and then subtracting that from 100 percent. This measurement is the amount of processor utilization

**PerformanceCounter(“Processor“, “% Interrupt Time“, “\_Total”);**

The rate, in average number of interrupts in incidents per second, at which the processor received and serviced hardware interrupts. It does not include deferred procedure calls, which are counted separately.

**PerformanceCounter(“Processor“, “% DPC Time“, “\_Total”);**

The percentage of time that the processor spent receiving and servicing deferred procedure calls during the sample interval. Deferred procedure calls are interrupts that run at a lower priority than standard interrupts.

**PerformanceCounter(“Processor“, “% Privileged Time“, “\_Total”);**

The percentage of non-idle processor time spent in privileged mode. Privileged mode is a processing mode designed for operating system components and hardware-manipulating drivers. It allows direct access to hardware and all memory. The alternative, user mode, is a restricted processing mode designed for applications, environment subsystems, and integral subsystems. The operating system switches application threads to privileged mode to gain access to operating system services. This includes time spent servicing interrupts and deferred procedure calls (DPCs). A high rate of privileged time might be caused by a large number of interrupts generated by a failing device. This counter displays the average busy time as a percentage of the sample time.

**2 CategoryName: Memory**

**PerformanceCounter(“Memory“, “Available MBytes“, null);**

This measures the amount of physical memory, in megabytes, available for running processes. If this value is less than 5 percent of the total physical RAM, that means there is insufficient memory, and that can increase paging activity.

**PerformanceCounter(“Memory“, “Committed Bytes“, null);**

it shows the amount of virtual memory, in bytes, that can be committed without having to extend the paging file(s). Committed memory is physical memory which has space reserved on the disk paging files. There can be one or more paging files on each physical drive. If the paging file(s) are expanded, this limit increases accordingly.

**PerformanceCounter(“Memory“, “Commit Limit“, null);**

it shows the amount of virtual memory, in bytes, that can be committed without having to extend the paging file(s). Committed memory is physical memory which has space reserved on the disk paging files. There can be one or more paging files on each physical drive. If the paging file(s) are expanded, this limit increases accordingly.

**PerformanceCounter(“Memory“, “% Committed Bytes In Use“, null);**

it shows the ratio of Memory\ Committed Bytes to the Memory\ Commit Limit. Committed memory is physical memory in use for which space has been reserved in the paging file so that it can be written to disk. The commit limit is determined by the size of the paging file. If the paging file is enlarged, the commit limit increases, and the ratio is reduced.

**PerformanceCounter(“Memory“, “Pool Paged Bytes“, null);**

it shows the size, in bytes, of the paged pool. Memory\ Pool Paged Bytes is calculated differently than Process\ Pool Paged Bytes, so it might not equal Process(\_Total )\ Pool Paged Bytes.

**PerformanceCounter(“Memory“, “Pool Nonpaged Bytes“, null);**

it shows the size, in bytes, of the nonpaged pool. Memory\ Pool Nonpaged Bytes is calculated differently than Process\ Pool Nonpaged Bytes, so it might not equal Process(\_Total )\ Pool Nonpaged Bytes.

**PerformanceCounter(“Memory“, “Cache Bytes“, null);**

it shows the sum of the values of System Cache Resident Bytes, System Driver Resident Bytes, System Code Resident Bytes, and Pool Paged Resident Bytes.

**3 CateroryName: PhysicalDisk**

**PerformanceCounter(“PhysicalDisk“, “Disk Read Bytes/sec“, “\_Total”);**

**PerformanceCounter(“PhysicalDisk“, “Disk Write Bytes/sec“, “\_Total”);**

it captures the total number of bytes sent to the disk (write) and retrieved from the disk (read) during write or read operations.

**PerformanceCounter(“PhysicalDisk“, “Avg. Disk sec/Read“, “\_Total”);**

**PerformanceCounter(“PhysicalDisk“, “Avg. Disk sec/Write“, “\_Total”);**

it captures the average time, in seconds, of a read/write of data from/to the disk.

**PerformanceCounter(“System“, “Context Switches/sec“, null);**

A context switch occurs when the kernel switches the processor from one thread to another—for example, when a thread with a higher priority than the running thread becomes ready. Context switching activity is important for several reasons. A program that monopolizes the processor lowers the rate of context switches because it does not allow much processor time for the other processes’ threads. A high rate of context switching means that the processor is being shared repeatedly—for example, by many threads of equal priority. A high context-switch rate often indicates that there are too many threads competing for the processors on the system. The System\Context Switches/sec counter reports systemwide context switches.

**4 CategoryName: Process**

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread’s set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread’s process. Threads can also have their own security context, which can be used for impersonating clients. (for more information  see “About Processes and Thread” http://msdn.microsoft.com/en-us/library/windows/desktop/ms681917%28v=vs.85%29.aspx)

**PerformanceCounter(“Process“, “Thread Count“, “\_Total”);**

The number of threads created by the process. This counter does not indicate which threads are busy and which are idle. It displays the last observed value, not an average.

**PerformanceCounter(“Process“, “Handle Count“, “\_Total”);**

the value reports the number of handles that processes opened for objects they create. Handles are used by programs to identify resources that they must access. The value of this counter tends to rise during a memory leak.

**PerformanceCounter(“Process“, “Thread Count“, “\_Total”);**

the value reports over time how many threads the processes create.

**5 CategoryName: System**

**PerformanceCounter(“System“, “Context Switches/sec“, null);**

A context switch occurs when the kernel switches the processor from one thread to another. A context switch might also occur when a thread with a higher priority than the running thread becomes ready or when a running thread must wait for some reason (such as an I/O operation). The Thread\Context Switches/sec counter value increases when the thread gets or loses the time of the processor.

**PerformanceCounter(“System“, “System Calls/sec“, null);**

This is the number of system calls being serviced by the CPU per second. By comparing the Processor’s Interrupts/sec with the System Calls/sec we can get a picture of how much effort the system requires to respond to attached hardware. On a healthy system, the Interrupts per second should be negligible in comparison to the number of System Calls per second. When the system has to repeatedly call interrupts for service, it’s indicative of a hardware failure.

**PerformanceCounter(“System“, “Processor Queue Length“, null);**

The System\Processor Queue Length counter is a rough indicator of the number of threads each processor is servicing.