TMurgent Technologies

White Paper
Processor Affinity

Multiple CPU Scheduling

November 3, 2003

Introduction[1]

The purpose of this paper is to examine the role of "processor affinity". In this paper I will explain:

- What processor affinity is.
- How the operating system uses processor affinity.
- Situations when manipulation of processor affinity has been used.

Finally, I will present TMurgent Technologies view on the appropriate and in-appropriate manipulation of processor affinity on Windows 2000 Server and Server 2003.

Before we begin a note of caution:

> Playing with processor affinity on a production system should never be done. We often find that otherwise intelligent human beings learn "just enough to be dangerous" and try to use that knowledge. We strongly recommend that first, you read the entire paper – especially our view on in-appropriate manipulation of processor affinity. Second, after you have done that, if you are still excited about playing with processor affinity please do it on a test system that is not in production.

[1] It is inevitable that a technical paper such as this will use terms that are Trademarks of other companies. Microsoft, Windows, SQL Server, IIS, and a variety of operating system names are trademarks of Microsoft Corporation. Citrix, MetaFrame, ICA, and SpeedScreen are trademarks of Citrix Systems. SysInternals is possibly a trademark of WinInternals. TMuLimit is a trademark of TMurgent Technologies.

I - What is “Processor Affinity”

In a multi-tasking operating system, the work to be performed is broken into discrete components, called tasks. These tasks are semi-independent portions of the total work, that may be scheduled to be executed when needed, then swapped out so that another task may run as appropriate. (This topic is discussed in more detail in our White Paper: *Scheduling Priorities, Everything you never wanted to know about OS Multi-Tasking* http://www.tmurgent.com/SchedulingWP.htm).

In the Windows Operating System, as in many modern multi-tasking OSs, the work to be performed is broken into *Processes* and further into *Threads*. A process is a complete unit that uses a virtual memory space. A process is usually (but no always) analogous to an application program. For example, executing Calc.exe results in a single process. Same with Paint.exe, or iexplore.exe. A process breaks down into one or more units called Threads. The threads of a process share the same virtual address space, but run semi-independently. Thus, while one thread is waiting for a file to be read-in from disk, another thread can be handling mouse and menu activity from the user. A thread is the unit that the operating system schedules to run from time to time.

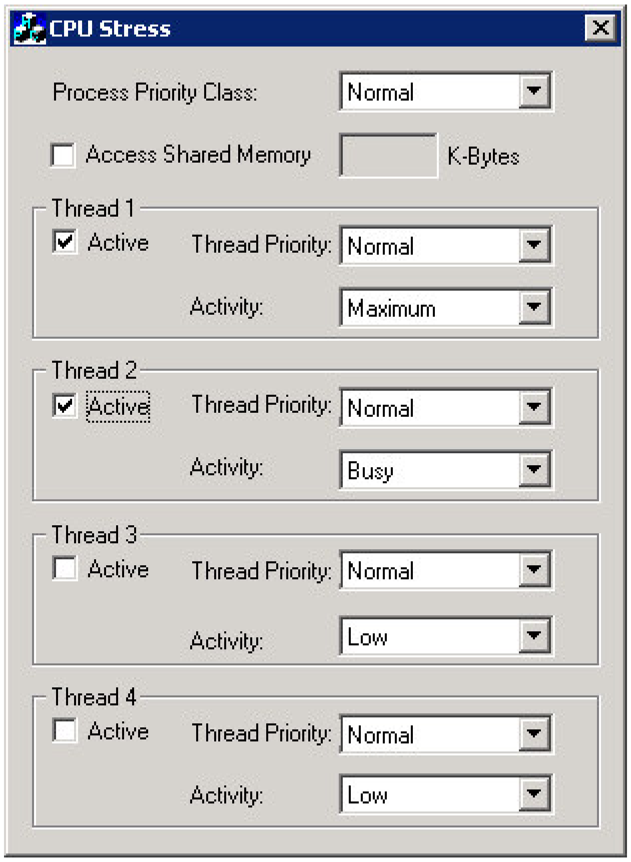
On a *Symmetric Multi Processor* (SMP) system, in popular use in servers today, the OS scheduler must not only decide when a thread can run, but where it should run. *Processor Affinity* is the term used for describing the rules for associating certain threads and certain processors. The term *Symmetric* in SMP indicates that any unit of software should be able to run on any processor – there is no “master processor”.

Asymmetric Multi Processor (ASMP)

In an *Asymmetric Multi Processing* System the processors are, or at least are treated, differently. If the hardware is not symmetric, this could mean that some processors do not have access to all memory, or all external devices. If the hardware is SMP capable, the operating system can implement ASMP. In this situation, typically the OS core runs constantly in one selected processor, with additional operating system and user tasks running in the remaining processor(s).

Windows NT, 2000, XP, and 2003 are all designed for SMP operation. Hardware vendors targeting this market develop to SMP as well (it is required to get the Microsoft sticker). As explained in Section II, some old device drivers may not be SMP aware (but not modern ones).

It is possible, even normal for some processes, for more than one thread of a process to executing at the same time. CPUStres.exe, from the *Windows 2000 Server Resource Toolkit*, makes a good example of this. If you want to try this, you can download the tools from the toolkit for free from the Microsoft TechNet site. CPUStres allows you to start one or more threads that will consume a complete CPU each. As shown in ***Figure 1 - CPUStres Example***, you can select the number of active threads (0 through 4) by checking the Active check-boxes as desired. If you set the Activity setting to an active thread to "Maximum", the thread will attempt to run all of the time. A single thread may ONLY be executing in a single processor at a time. Used on an N-way system (a system with "N" processors), each thread will consume close to (100/N) percent CPU. Thus the settings shown would consume 100% of a dual processor system, with one of the two threads pretty much constantly running in each processor.


CPU Stress

Process Priority Class: Normal

☐ Access Shared Memory K-Bytes

Thread 1
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 2
☑ Active Thread Priority: Normal
Activity: Busy

Thread 3
☐ Active Thread Priority: Normal
Activity: Low

Thread 4
☐ Active Thread Priority: Normal
Activity: Low

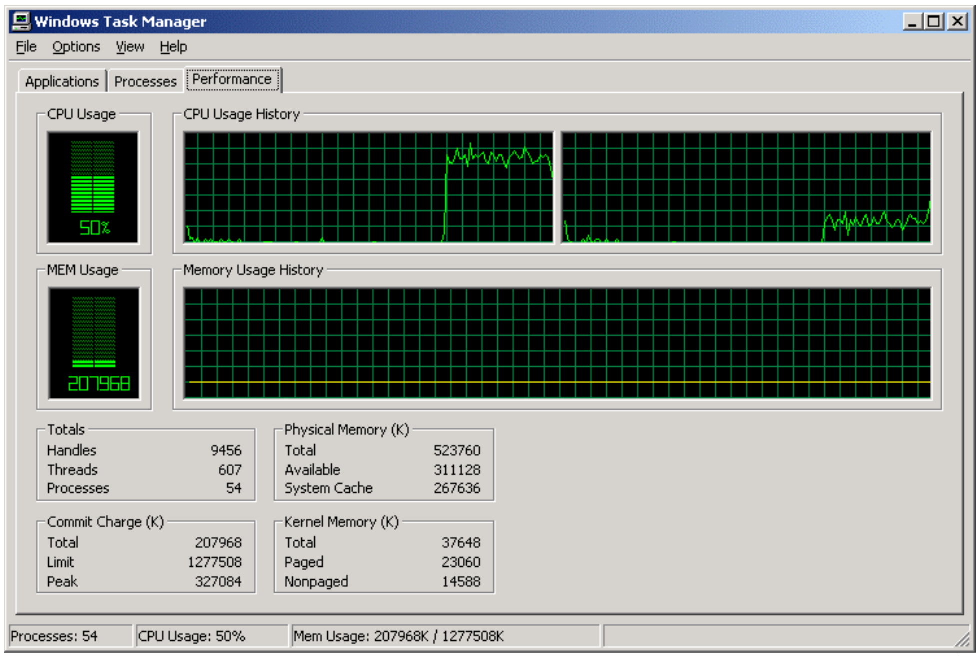
Figure 1 - CPUStres Example

The operating system uses *Processor Affinity* when the OS task scheduler assigns threads to run in processors. As we will see, this affinity may be a hard set rule (thread A may only run in processor #3) or it may be a preference (thread A should try to be scheduled into processor #3 first). By default each process/thread has affinity for every processor in the system.

II - How the Operating System Uses Processor Affinity

As mentioned earlier, the Windows Server Operating Systems are all SMP operating systems. So, at least in theory every thread may be scheduled into any processor at any time. Let's look at what happens if we try to run a single thread constantly on an otherwise calm system.

Figure 2 shows a view of the Task Manager on a dual processor system. As can be seen, the system was initially rather idle, after which we activated one thread at the maximum level. The total CPU consumed was 50% (shown on the left),



Totals	
Handles	9456
Threads	607
Processes	54

Physical Memory (K)	
Total	523760
Available	311128
System Cache	267636

Commit Charge (K)	
Total	207968
Limit	1277508
Peak	327084

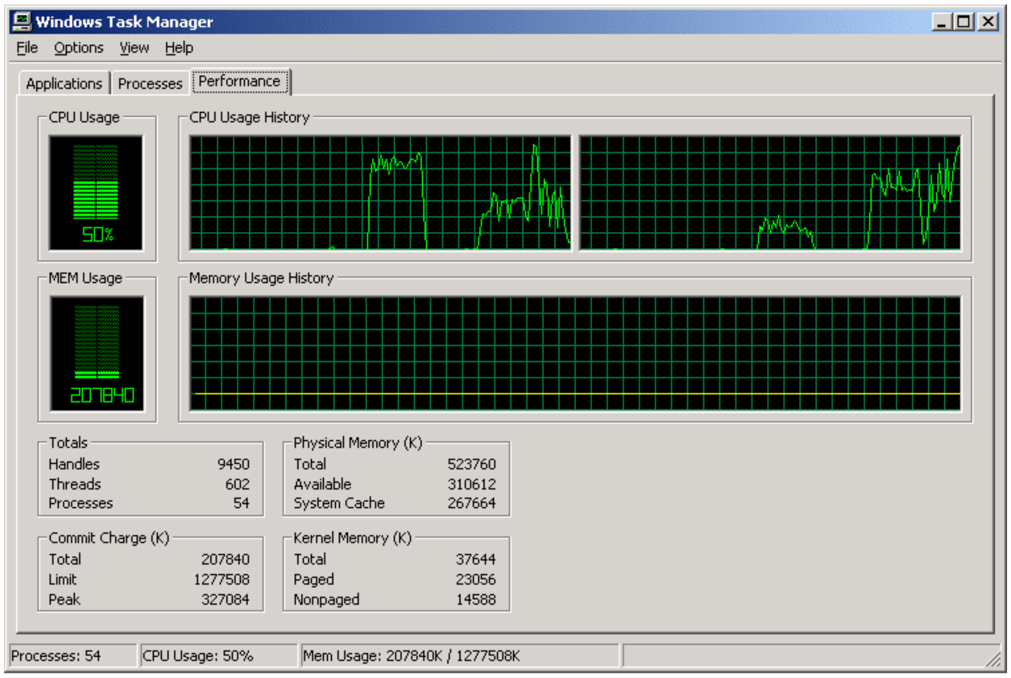
Kernel Memory (K)	
Total	37648
Paged	23060
Nonpaged	14588

Processes: 54 | CPU Usage: 50% | Mem Usage: 207968K / 1277508K

Figure 2 - Scheduling on one busy thread

which is as expected (100/2 = 50). Looking at the CPU History Graph is revealing. This graph is showing CPU activity separately for each processor. As shown, the two CPU used different amounts of CPU.

In ***Figure 3***, we ran the same test again a few moments later. Here we see the



50%

207840

Totals	
Handles	9450
Threads	602
Processes	54

Physical Memory (K)	
Total	523760
Available	310612
System Cache	267664

Commit Charge (K)	
Total	207840
Limit	1277508
Peak	327084

Kernel Memory (K)	
Total	37644
Paged	23056
Nonpaged	14588

Processes: 54 CPU Usage: 50% Mem Usage: 207840K / 1277508K

Figure 3 - Scheduling one busy thread (again)

total is 50%, however this time the scheduler placed the thread into the second processor more often. What accounts for the behavior shown?

If you expected to see one of the processors used 100% and the other at roughly 0%, this is explained through normal thread scheduling. First, we must realize that no thread is allowed to just constantly run. Threads normally run for a short while and then voluntarily yield. The yield may be because it is sleeping on a timer, waiting for input from the user, waiting for a file to read in, or waiting for access to a shared resource (such as a shared memory lock). CPUstres, when set for maximum does not yield voluntarily, but is caught by the system if it does not voluntarily yield within a given time frame (a value determined from the quantum limit table, but roughly 60-240ms).

Even if the system is at a 0% load, there are threads running. Very briefly some system threads will wake up and do a very small amount of processing. Even when there are none of those, there still is the “System Idle Thread” (or “Zero Page Thread” as it is sometimes referred). The System Idle Thread is scheduled by the scheduler whenever there is no other task to run in a processor. Several important housekeeping activities take place there. When the scheduler tries to schedule a normal process and one processor is busy while the other is free (the System Idle Thread is running), and no special processor affinity has been requested, it will schedule the task into the free processor.

This explains why it is not a 100/0 split, but fails to explain why it isn't at least near to a 50/50 split, as in random placement. To understand why, we need to introduce the concept of a preferred processor.

II.1 Preferred Processor

The Windows task scheduler uses a mechanism called the preferred processor. Each thread that is not currently running in a processor has a "preferred processor". The scheduler uses this as a form of processor affinity when a thread is ready to be scheduled into. The scheduler will schedule the thread into its preferred processor if that processor is free. If that processor is not free, it will schedule the process into any other free processor. This is done for performance reasons (see sidebar). The preferred processor of a thread is the processor it last ran in. So on a system that is not overly busy, threads tend to be scheduled into the same processor more often than random occurrence.

Preferred Processor

In an SMP system, although each processor has access to all memory, each processor has its own cache copy of certain portions of memory (There are both Layer 1 and Layer 2 caches, typically). When the processor can retrieve code or data from that cache instead of going over the memory bus to slower DRAM based main memory, it will perform faster.

The scheduler uses this Preferred Processor (also called Ideal Processor) method of affinity to increase the odds of cache hits to improve performance.

There are implications on this for Hyper-Threaded processors with Windows Server 2003, where pairs of logical processors share a physical processor – and a single L1/L2 cache. We haven't found proof on this yet, but we believe the 2003 scheduler will try to schedule to the preferred processor first, and then try any logical processor on the same physical processor.

Of course, the timing of other activities can affect the likelihood of a preferred processor miss. When a preferred processor miss occurs, the preferred processor for the thread is reset to the processor it is scheduled into next.

Non Uniform Memory Access (NUMA)

A *Non Uniform Memory Access* System uses a design that has multiple bus/memory subsystems in an attempt to reduce the bottleneck of a single bus/memory system. In a NUMA design, every processor has access to all memory; however, each processor has faster access to a portion of the memory, and slower access to the rest. The difference in speed is normally no more than 1:3. While NT and 2000 are not "NUMA aware" (but will run non optimally), Server 2003 is NUMA aware.

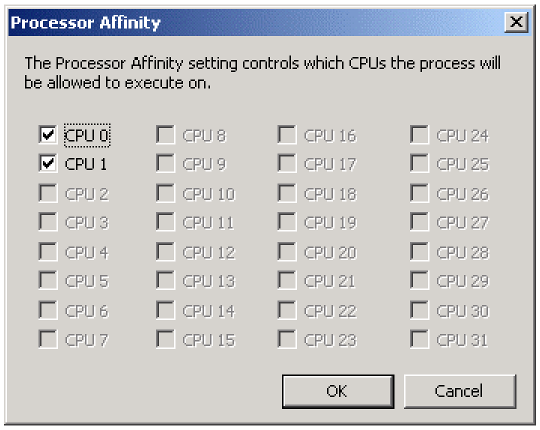
We will not get into NUMA in detail here since it is only in use in very high end systems, however the 2003 scheduler is aware as to what CPUs are in which nodes and will try to first schedule into the preferred processor, then into any processor that is in the same NUMA node .

By the way, the OS will never try to move a currently running thread out of a processor so that a different thread with that as the preferred processor may run in that processor. Nor will a thread wait until it's preferred processor becomes free. It will be scheduled into any free processor available. You can think of the preferred processor as a kind of a hint to the OS scheduler. If all else is even, try to put it here.

11.2 Process Affinity via Task Manager

The Windows Task Manager, as well as third party tools from folks such as SysInternals (www.sysinternals.com) allow you to view and modify with Affinity on a process level. To view process affinity in the Task Manager, you need to be on a multi-processor machine. Click on the "Processes" tab to expose the list of processes running in the system. If you right click on a line in the list, a pop-up menu appears. One of the items in that menu on a multi-processor system is "Set Affinity". If you select a user process you will be presented with a dialog similar to that of ***Figure 4*** . Note that even logged in with administrative privileges, one cannot set (or view) the processor affinity of system processes – it must be a user level task. In our case, we selected our friend CPUStres.

The display shows a checkbox for each possible CPU. Note that the number of boxes shown that are disabled may depend upon the number of CPU the version of the OS is licensed for. In this case, we are running Windows 2000 Advanced Server which supports up to 32 processors. We have two installed, thus CPU 0 and CPU 1 are active checkboxes.



Processor Affinity

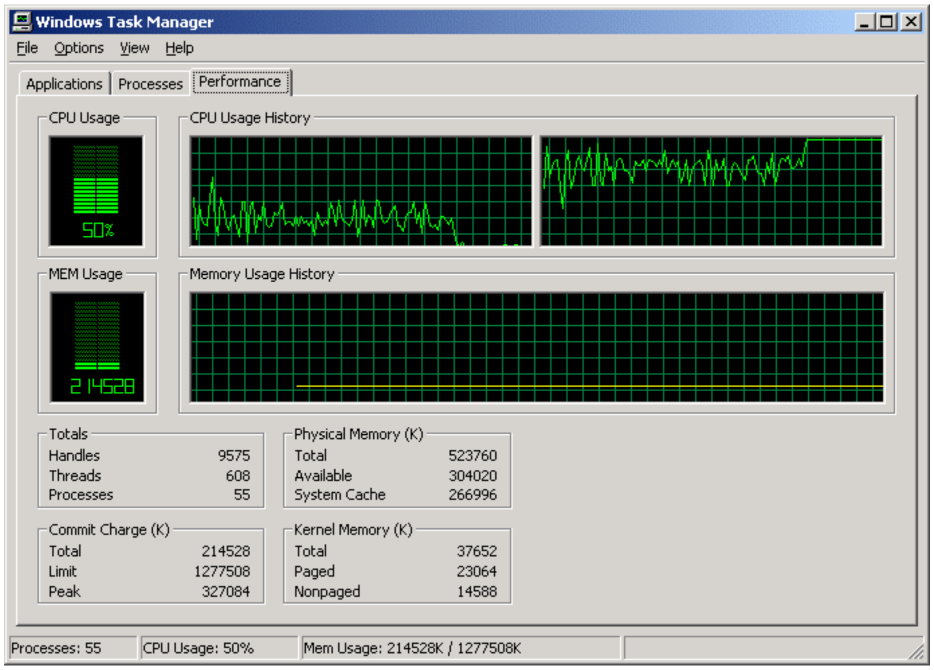
The Processor Affinity setting controls which CPUs the process will be allowed to execute on.

✓ CPU 0	☐ CPU 8	☐ CPU 16	☐ CPU 24
✓ CPU 1	☐ CPU 9	☐ CPU 17	☐ CPU 25
☐ CPU 2	☐ CPU 10	☐ CPU 18	☐ CPU 26
☐ CPU 3	☐ CPU 11	☐ CPU 19	☐ CPU 27
☐ CPU 4	☐ CPU 12	☐ CPU 20	☐ CPU 28
☐ CPU 5	☐ CPU 13	☐ CPU 21	☐ CPU 29
☐ CPU 6	☐ CPU 14	☐ CPU 22	☐ CPU 30
☐ CPU 7	☐ CPU 15	☐ CPU 23	☐ CPU 31

Figure 4 - Process Affinity in Task Manager

By default, all processes may run in any processor. This includes user as well as system processes. The task manager does not allow you to modify the process affinity of system processes to ensure system stability. If high priority system threads are prevented from running due to process affinity, the system could lock up.

We chose to clear the checkbox on CPU 0 for the CPUStres process, allowing it to run threads in CPU 1 only. The results are shown in ***Figure 5***. As soon as we set the processor level affinity, the first processor (CPU 0) became almost idle, while the second processor (CPU 1) became 100% busy. Because the system is an SMP system without any non-SMP drivers, we could have just as easily selected affinity for CPU 0.



```
Totals                      Physical Memory (K)
Handles          9575       Total            523760
Threads           608       Available        304020
Processes          55       System Cache     266996

Commit Charge (K)           Kernel Memory (K)
Total          214528       Total             37652
Limit         1277508       Paged             23064
Peak           327084       Nonpaged          14588

Processes: 55   CPU Usage: 50%   Mem Usage: 214528K / 1277508K
```

Figure 5 - CPUStres with Processor Affinity

While fun to play with, the Windows Task Manager is rather limited. It is a run-time only tool, so while you can change a currently running process, you cannot use it to set the affinity of an application every time it runs. To do that you need some add-on software (see sidebar).

Imagecfg.exe

Microsoft does have a tool for permanently setting the processor affinity associated with an executable each time it is run. Although not in the original Windows 2000 Server Resource Toolkit, there is a supplement, called "Windows 2000 Server Resource Kit Supplement One". The tool does not appear to be included in the Microsoft downloads anywhere, so you probably need to purchase the supplement.

To set processor affinity, you run imagecfg.exe with the –a option, followed by an affinity mask (a hexadecimal formatted number) and the fully qualified name of the executable. The bit mask is starts with CPU0 as bit one, CPU1 as bit two, etc. For example:

```
        Imagecfg –a 0x03 c:\tools\iexplore.exe
```

would allow iexplore to run on either CPU0 or CPU1.

The tool does not work on 16-bit executables and SHOULD NOT EVER be used on kernel components. Although in the 2000 kit, the tool works on NT/TSE as well.

The Windows Task Manager also only works on affinity of a process basis. This is convenient, as any new threads that start after you set the affinity will inherit the process affinity setting. However it lacks the flexibility of setting affinity on a per thread basis.

For example, take our friend CPUStres. The process consists of a thread that runs the GUI (Graphical User Interface), plus a thread for each activated checkbox. In our situation depicted in ***Figure 5***, the process consists of the GUI thread and one CPU hog thread. The GUI thread must now compete for cycles with the thread that is consuming all the CPU. In this example, using the mouse on the GUI will still respond quickly because the GUI thread will be a higher priority than the CPU hog thread. So while this is OK for this example – it is a very simple example. Herein lies the conundrum with manually assigning processor affinity. If you only consider that these two threads may be running you may believe you understand the best solution. But what about the other threads running in the system? Even in our idle server with no terminal server sessions running we have over 600 threads. A better set-up, if you need to assign affinity, would be to only assign processor affinity to the CPU hog thread, and let the GUI thread roam to any available processor.

CPUStres Design Note

The design of the application is that there is one thread for the GUI, and one for each active thread checkbox. If none of the checkboxes are checked a second thread is created that is idle.

You can see the number of threads present in a task by adding the Threads column to the Process Tab display of the Task Manager. Select the menu View→Select Columns to add the column to the display.

In a White Paper we released earlier this year, *Perceived Performance* (www.tmurgent.com/PerceivedPerformance.pdf), we explain how the scalability of a multi-user system is dictated by the performance perceived by the users. That perception is guided by the responsiveness of mouse and keyboard actions. We may be getting ahead of ourselves here, but none of us are really smart enough to plan out an optimal fixed affinity plan for a server that works under the varying workloads that will be presented.

Also keep in mind that while affinity can be used to keep a user process from using certain processors, it cannot be used to guarantee a process exclusive use of a processor. That is because of all those system processes that must have affinity for all processors installed. These system processes will interrupt the application you are trying to dedicate to the processor.

III - Situations Where Processor Affinity Has Been Used.

We can talk about four situations where fixed processor affinity has been used. There no doubt have been others, however these two represent the “reasonable” use of fixed processor affinity.

III.1 Legacy Device Drivers

Processor Affinity was necessary in the past with some device drivers, especially on Windows NT. These drivers were generally written so as not to be SMP aware, and locking them into a specific processor was a reasonable work-around to get them functional. Typically, interrupts would be configured to only interrupt a single processor (sometimes called Interrupt Affinity).

The lore of affinity was propagated over the years, especially in relation to LAN card drivers, to a belief by some that the OS uses the first processor and that these drivers should have affinity set to use the last processor.

While there may have been specific drivers that needed to be tied to the last processor, it certainly was not because the OS needed to run in the first processor. Such ideas fit in with having to wait two hours after eating before swimming. (My mother insisted on such a belief for several years after we moved to a house on a lake. Thank goodness she finally wised up!)

In any case, drivers that run on Windows 2000 and 2003 should not have affinity issues. Certainly any driver that is certified (received Microsoft certification for either OS) cannot have an affinity requirement. Microsoft has been very clear on the requirements, and the driver must be able to execute in any processor.

Certainly from a performance perspective, you want the driver processing to happen in as timely as fashion as possible. While the interrupt will occur nearly immediately in any case (only a higher priority interrupt can prevent the driver interrupt from immediately responding), modern driver software only executes a minimal set of instructions from within the Interrupt Service Routine (ISR). Most of the processing is deferred into a Deferred Processing Call (DPC). The DPC is executed when a current running task in one of the allowed processors completes its execution (i.e. either it voluntarily yields, or exceeds its quantum limit). Clearly the more processors that are available to be selected, the less delay in completing the processing of the interrupt request.

III.2 Single Process Anomaly

A second case for the use of fixed process affinity is as a cure for what I call the single process anomaly. An example is a customer that runs a terminal server for a number of users that are running the typical business applications – the office suite, internet explorer, etc. They have one user, an engineer, which needs to use a popular CAD package (application name withheld because it isn't the vendors fault. Any CAD half-decent package would act the same way). Being very much into thin client computing it is necessary to run the CAD application on the server, since it is the only device with the memory and horsepower required.

When the user runs the application, however, the package uses multiple threads simultaneously and even on a quad processor it will at times consume 100% of each processor. Needless to say, the other users are not amused. By using fixed processor affinity, you can restrict the CAD app threads to only run in one or two of the processors, leaving the remaining processors to handle the needs of the other users.

There is a saying that goes: "If all you have is a hammer, everything starts looking like a nail". If fixed processor affinity is all you have as your tool, it does work OK. It is not, however, the best tool for the job. Let us look at an example.

In this example, we will assume we have a quad-processor system. The administrator implementing a fixed affinity solution only has four choices, 4 processors (the default), 3, 2, and 1. By assigning the application an affinity of one processor (for this example we will pick CPU 2), we ensure that at all times a minimum of 75% of the processing power is available to the other system and user processes. Meanwhile the CAD application is limited to a maximum of 25% of the processing power. The CAD app still must compete with other threads for that 25%. When the OS thread scheduler tries to schedule the CAD app, there must be no equal or higher priority thread currently running in that processor. The scheduler will not move a thread running in CPU 2 to another processor, even if there is an idle processor and the ready CAD app has to wait.

A better solution would be to use the SuperMax capability of our TMuLimit product. The Administrator would have the flexibility of assigning a maximum CPU utilization at any percentage level, not just multiples of 100/N. Because the application is not restricted to a single CPU, it could also avoid being held ready when free processors are available for use. Finally, the solution

About TMuLimit

TMuLimit, implemented as a service, monitors and controls applications automatically on a real-time basis. While most applications are improved using the Priority Management, applications that consume unrestrained amounts of CPU, like CPUstres (or any 16 bit application) are better handled by a method we developed to override the OS thread scheduler. We called this method SuperMax.

You can find more information on TMuLimit at http://www.tmurgent.com/TMuLimit.htm scales when suddenly a second engineer needs access to the application. By the way, TMuLimit does also provide support for Affinity assignment, however we strongly recommend against using it.

III.3 The ICA Administrator Toolbar

In some installations of Citrix MetaFrame, the icabar (ICA Administrator Toolbar) is set for fixed processor affinity. This is the toolbar that shows up on the right side of the Administrator's desktop (until the administrator disables it) with icon shortcuts to the standard administrator programs associated with MetaFrame, such as SpeedScreen or the ICA Management Console.

I cannot fathom a reason to assign affinity to such a benign tool, but there it is. Because the tool never uses much CPU, I guess it does no harm either. Even if the first thing you do is not to disable the toolbar.

III.4 SQL Server/IIS

Administrators configuring Microsoft SQL Server will sometimes use fixed processor affinity. The use is primarily been on systems that are also loaded with additional processing requirements. As explained in "Administering SQL Server" (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/adminsql/ad_config_6rw2.asp) this is not a recommended practice.

IIS 6.0 actually includes an optional configuration setting to apply processor affinity to worker threads. Given the large number of worker threads, we are doubtful of the effectiveness of such a setting (but we have not tested it).

IV - TMurgent Technologies View on Processor Affinity.

We sometimes hear from customers that become interested in Affinity Management. I have to admit being briefly infatuated with affinity for a short period at one time. But as explained in this paper, especially in the situations in ***Section III*** of this paper, we recommend against using processor affinity.

If you still have drivers that require affinity, they are likely also destabilizing your system in other ways. Such old drivers should be replaced at any cost.

Affinity can be used to limit a process, or guarantee other processes, certain CPU levels. But it is not the best tool for the job. Such a solution sacrifices flexibility, responsiveness, and scalability.

And finally, a fixed processor affinity solution cannot take into account the different system conditions that will exist at different times. Process management requires both measurement of how the system is behaving and real-time adjustments in order to generate optimal peak performance.

Preferred Processor Affinity done by the Operating System does a somewhat reasonable job of optimizing for processor specific cache. Assuming that Server 2003 schedules with a logical processor aware methodology such as we suggested in the side note (see **Preferred Processor**) we also have the optimum in the Hyper-Threaded and NUMA arenas.

Not that the OS can't do better. Modifications to the basic scheduler to give a small chance for a preferred processor to free up (like maybe for a half a millisecond) before sending it to a different physical processor *may* provide improved results. We will need a lot of testing to optimize that idea out!

While we are on the subject of improving the system, a couple of other design notes. We fundamentally believe that the HAL clock quantum (10-15ms) is far too slow for the processor speeds today. This hasn't changed since before a 600mhz uni-processor was a screaming machine. Time for a change here folks! Also, the OS performs some background functions, such as managing the free memory pool only once a second. It is probably time to speed that up on high-end systems as well.

Conclusion

In this paper, we described processor affinity and how it is used. We show some simple tools that you can use to view and manipulate affinity. We also describe some situations in which affinity has been used in the past.

Although we hint to it earlier in the paper, finally, we provided our recommendation on the place of affinity. Our purpose is to guide the reader to use their head for what is important to their system, rather than jumping on a bandwagon of something that “sounds cool”.

As always, we strongly recommend that customers thoroughly test any ideas, software, or configuration changes initially on test systems before introducing changes to a production system.

References and useful links

The following are useful references, some of which were used in the development of this White Paper.

Ref 1 ***Scheduling Priorities: Everything you never wanted to know about OS Multi-tasking***, TMurgent Technologies, July 2003, **http://www.tmurgent.com/SchedulingWP.htm**

Ref 2 **SysInternals Web Site.** SysInternals provides a wonderful variety of tools and articles to investigate the Windows Operating System. **http://www.sysinternals.com**

Ref 3 ***Perceived Performance: Tuning a System for What Really Matters,*** TMurgent Technologies, September 2003. **http://www.tmurgent.com/images/perceivedperformance.pdf**

Ref 4 ***Administering SQL Server,*** Microsoft Corporation**.** **http://msdn.microsoft.com/library/default.asp?url=/library/en-us/adminsql/ad_config_6rw2.asp**