Inside Windows NT SMP Scheduling

last updated March 18, 1997

Flaw in NT SMP Support Could Hinder Scalability

Note: The information presented here is the result of my own study. I have no access to NT 4.0 source code.

Introduction

Windows NT was designed from the start to support symmetric multiprocessing (SMP) execution. As multiprocessors have become more common-place however, NT has gotten a reputation of not scaling well as the number of processors increases. As more CPUs are added, the increase in performance rapidly declines. One way operating systems like NT try to make themselves SMP-friendly is to use *affinity* heuristics in their scheduling algorithms. Given a choice of threads to run on a CPU, they will try to run one that has recently been scheduled on that CPU, with the hope that part of the thread's data is still present in the CPU's on-chip cache.

This article will describe NT's use of affinity in its SMP scheduling algorithms. In addition, I'll present the algorithm NT uses to pick which CPU a thread should run on. I'll show how the algorithm is flawed in one significant way, involving improper use of affinity, how the flaw can lead to unnecessary thread migration across processors, and how it could be a factor in preventing NT from scaling well on SMPs.

NT Scheduling Basics

The NT scheduler is a very simple one. A thread can have one of 32 priorities, 0-31, with higher numbers corresponding to higher priorities. Priorities from 0 to 15 are dynamic priorities, because the system can temporarily change the priorities of threads in this range when certain events take place. The 16 to 31 range is called the "real-time" range. Real-time threads execute at fixed, unchanging priorities. The goal of the NT scheduler on an n-processor system is to always be executing the n highest priority threads that are ready for execution.

Processors on an SMP system are numbered from 0 to n-1. The numbering is only used for the purposes of identification, and is not priority-related.

Scheduling Decisions

There are two situations when NT's scheduler is invoked. The first is when a thread's turn on a CPU is ended in one of the following ways:

- it blocks (waits) for some event to take place
- it terminates
- it yields its turn
- its turn (quantum) ends

The NT scheduler must decide which thread of the threads that are ready to run it should schedule next on the CPU. Factors that can affect its choice include which threads have hard-affinity for the CPU, and which threads have soft-affinity for the CPU. Hard-affinity describes a thread's ability, as specified by applications, to run on certain CPUs and not on others. By default a thread can execute on any processor. Soft-affinity is the name given to a scheduler's attempt to keep a thread executing on the same processor in the hope that some of its data stored in the processor's cache the previous time it ran will still be present.

The scheduler is also necessary to handle the transition of a thread from a waiting state to a ready-to-run state. For example, a thread might block on a timer so that it will wait until an interval has expired. When the timer triggers the thread is ready to continue. The system could wait until another one of the executing thread's ends its turn in one of the above listed ways. Instead, NT determines if the thread that has become ready to execute should preempt (replace) a lower priority executing thread.

Picking the Next Thread

NT maintains a list for each priority level. These lists are where threads are placed that are ready to execute, but not scheduled on a CPU. A 32-bit mask is also maintained so that NT can quickly determine which priority levels have entries in their lists. An 'on' bit in the mask means that the list for the priority corresponding to the bit's position is not empty.

The function in the scheduler that is called when a new thread must be picked to execute is called **KiFindReadyThread**. The algorithm it implements proceeds as follows:

1. Determine the highest priority level with ready threads
2. Consider the next thread in the list. If there are no more entries on the list, move to the next lower priority level's list and consider the first thread on it.
3. If the thread's hard affinity prevents it from running on this CPU goto 2.
4. Mark this thread as the primary candidate
5. If the thread last executed on this CPU, its been more than 20ms since the thread last ran, or the thread has a priority above 24 (high real-time), schedule this thread.
6. Otherwise, scan the list for another thread that last executed on this CPU or that hasn't run in more than 20ms. If one is found, schedule it instead. If one is not found, schedule the primary candidate.

The algorithm is illustrated with the following example of a 2-processor SMP system with CPUs numbered 0 and 1. CPU 0's system timer interrupts and the quantum tracking code determines that the currently executing thread's quantum has expired. The thread is executing at priority 8 and the dispatcher queues look like Figure 1, below.

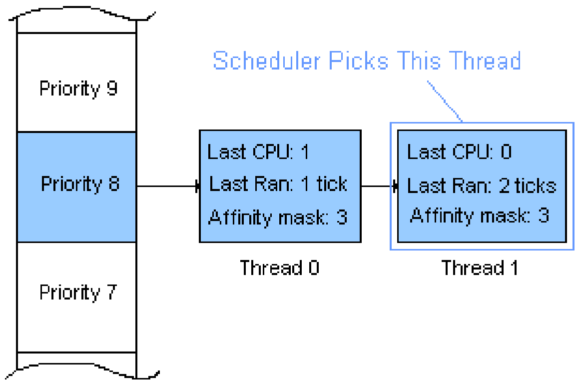

Figure 1. KiFindReadyThread finding the next thread to schedule

The scheduler starts with the first thread on the priority 8 queue, which is thread 0. The thread's affinity mask, which is a bitmask of the CPUs the thread can run on, indicates by having bits 0 and 1 "on" that the thread can run on either processor. Since the thread's priority is not above 24,. it has not been more than 20ms since the thread last ran, and the thread didn't run on CPU 0 the last time it ran, the scheduler marks it as the primary candidate, but moves on to examine the other threads in the list. When it looks at thread 1, it sees that one of the two conditions that would result in it becoming the primary candidate holds. The thread last ran on the CPU on which the scheduler is making its decision, so soft-affinity dictates that it will be the chosen thread. Therefore, the scheduler discards its original choice and makes thread 1 the next thread to execute.

This algorithm honors hard-affinity, while attempting to schedule one of the highest priority threads that last ran on the given CPU (soft-affinity), or that have not run in more than 20ms.

Should this Thread Execute?

When a thread becomes ready to execute, NT must decide if the thread should preempt one of those currently running on a CPU. This is because NT's goal is to run the highest priority threads in the system. The subroutine that performs the decision is called **KiReadyThread**. Its algorithm is listed below. In the description, *ready thread* refers to the thread that has become ready to execute.

1. If there are one or more idle processors, schedule the ready thread on current CPU if its idle, or the one with the highest number.
2. Otherwise, if the thread executing on the processor that the ready thread last ran on has a lower priority than the ready thread, schedule the ready thread on that processor. This is the flaw that can cause unnecessary thread migrations.
3. If no processor was found in the above steps, place the thread at the front of the ready list associated with its priority level if it was preempted, and at the end if it was not. The thread will be scheduled by **KiFindReadyThread** when a subsequent thread turn ends. This is not good for real-time applications, and can also cause unnecessary thread migrations.

When a thread is scheduled to take over a processor, it is placed in standby-mode on that processor. If it is not the same processor on which the scheduler is executing, NT must inform the processor that it should swap threads. It does so by sending an *inter-processor interrupt* (IPI). When a CPU receives an IPI it checks to see if a thread has been placed in standby-mode and if so, swaps it onto the CPU and executes the **KiReadyThread** algorithm on the preempted thread.

This algorithm is demonstrated with Figure 2. The system has 4 processors with threads of the specified priorities executing on each one. A thread becomes ready to execute because it was waiting on an event that one of the active threads has signaled.

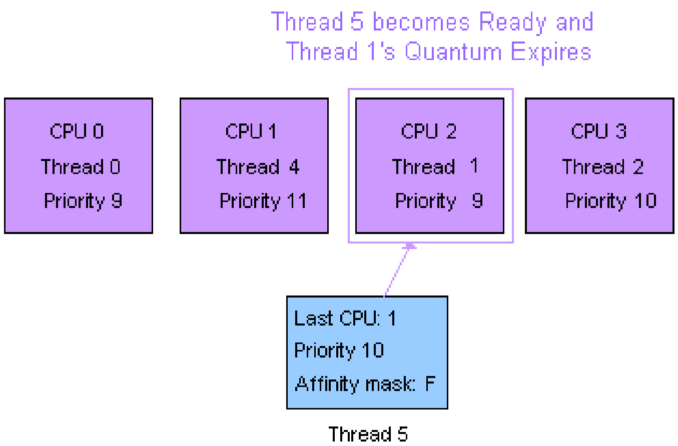

Figure 2. The scheduler finding a processor for thread 5

The scheduler first looks at the CPU thread 5 last ran on, CPU 1, and sees if the thread executing on it has a lower priority. Thread 4 has a priority higher than thread 5, so the scheduler places thread 5 on the scheduler's priority 10 ready list. Soon after, the quantum of thread 1 expires on CPU 2. **KiFindReadyThread** in invoked to find the highest priority ready thread to execute, and it finds thread 5.

The scheduler preempts thread 1 and calls **KiReadyThread** on it, which results in its being placed in the ready list for priority 9.

The Flaw The algorithm employed by **KiReadyThread** contains a flaw which can cause unnecessary overhead when a SMP system. The flaw is in Step 2 where the scheduler will run the ready thread on the last processor it executed on if the thread has a higher priority than the thread currently executing on that processor. It will do so regardless of how long its been since that the thread last ran; it could have been hours. A second place where unnecessary overhead can be caused is Step 3. In that step the scheduler gives up preempting another thread with the ready thread and just places the ready thread in the scheduler's ready lists.

Why is Step 2 a flaw and how can Step 3 cause extra thread migration? Lets look at a worst-case scenario, shown below in Figure 3.

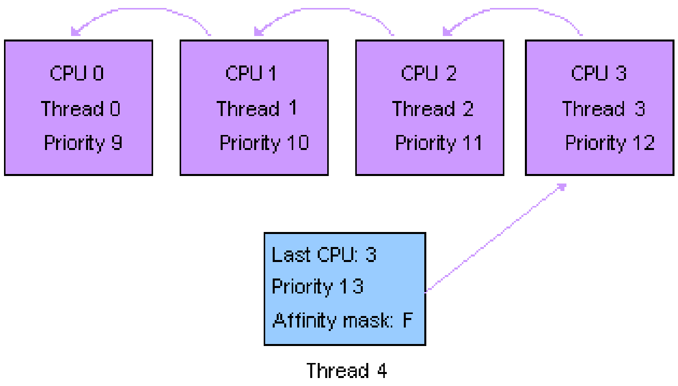

Figure 3. Worst-case thread migration on a 4-way SMP system

This 4-processor system is running the threads shown. At some point thread 4 becomes ready to execute and **KiReadyThread** is invoked, which leads to the activation of the Step 2 flaw. The algorithm starts by examining the thread executing on the CPU that thread 4 last executed on, CPU 3. The thread there, thread 3, has a priority lower than thread 4's, so the scheduler places thread 4 in standby on CPU 3 and sends CPU 3 an IPI.

CPU 3, upon receiving the IPI, preempts thread 3, schedules thread 4, and calls **KiReadyThread** with thread 3 because it is still ready to execute. Not being able to place thread 3 on the last CPU it ran on, CPU 3, because thread 4 has higher priority, the scheduler places the thread in the ready lists. Until a scheduling event like a thread exit, a quantum expiration, or a thread block occurs, threads with a lower priority than a thread in the ready lists will be executing. This makes it impossible for NT to provide real-time guarantees to threads on a multiprocessor system. In addition, during this time the NT scheduling rule of running the *n* highest priority threads on an *n*-way SMP is violated.

Unnecessary thread migration is also caused by Step 3 and the first occurs when the thread running on CPU 2 has its turn end. Here, threads are migrated because the scheduler ignores global priorities. The scheduler looks for a new thread for CPU 2. **KiFindReadyThread** finds thread 3 in the ready lists so it places it on the processor and puts thread 2 in the ready lists. CPU 1's thread quantum expires next and the scheduler preempts thread 1 in favor of thread 2. Finally, thread 0's turn on processor 0 ends and **KiFindReadyThread** replaces it with thread 1.

This example shows how the algorithm causes 3 unnecessary context switches: 1 from the flaw in Step 2 of the **KiReadyThread** algorithm and 2 from Step 3. These 3 switches could be avoided if

KiReadyThread found the lowest priority executing thread (in this case thread 0), and replaced it with the higher-priority ready thread if possible.

The situation is made worse when there are high priority threads that are bound to each processor by the algorithm's use of soft-affinity and they periodically become ready to do work, causing a ping-pong effect on lower priority threads. We'll see this effect in an example.

A context switch results in the following actions, each of which carries with it overheads:

- the old processor state must be saved and the new one restored. This may include floating point/MMX state as well.
- the newly scheduled thread will begin executing with a cold cache (none of the data it needs will be in the on-chip cache).
- the system's page tables will be reloaded and the processor's virtual memory Translation Lookaside Table (TLB) will be flushed if the newly scheduled thread belongs to a process different than the one being switched out.

Thus, it is in the interest of performance and scalability that the scheduler should try and minimize context switches, and therefore thread migrations.

An Example

The flaw of Step 2 in the algorithm is graphically visible in the *Performance Monitor* screen shot displayed in Figure 4. The picture shows a two processor system running the *Cpuhog* program, but that is otherwise idle. *Cpuhog* is a program with a single thread that executes at priority 15 and sits in an infinite busy loop. Processor utilization is shown along the left side of the graph, and the utilization of each processor is assigned a separate color. 100 seconds of time-window is pictured lengthwise.

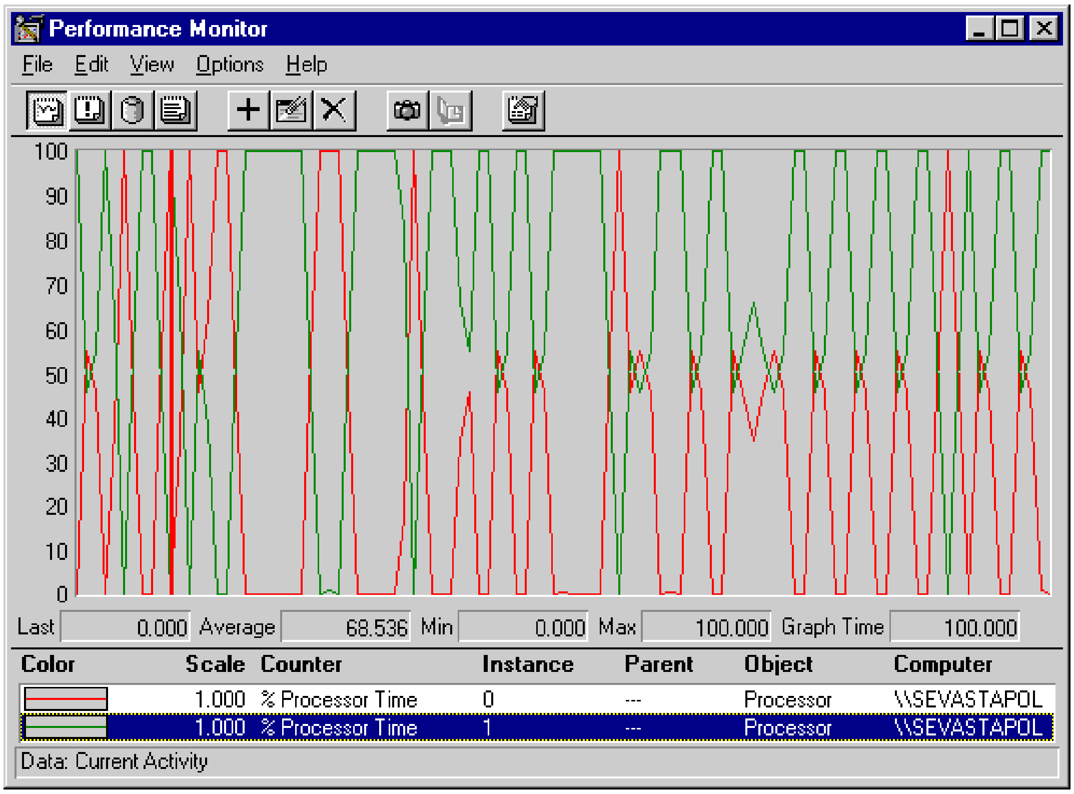

Figure 4. Cpuhog being forced to migrate from processor to processor

Because *Cpuhog* consumes all available CPU cycles in its loop, the processor it executes on will be 100% utilized. Everywhere the colored lines criss-cross *Cpuhog* has migrated from one processor to the other. In this particular example, the migration is caused by system worker threads becoming ready to execute. System worker threads perform system activity and execute at priority 16. Most of the time the work items are short in duration, taking no more than a few milliseconds to complete. When a system thread becomes ready to execute, **KiReadyThread** examines the thread executing on the processor the system thread last ran on, and unless another system thread of equal or higher priority happens to be executing on it, the system thread will take over the CPU. For example, say *Cpuhog* is executing on processor 0 and *Explorer* (priority 8) is executing on processor 1, and a system thread that last ran on processor 0 becomes ready. *Cpuhog* will be forced off the CPU and be rescheduled on processor 1. This results in an unnecessary context switch. This sequence of events can be seen quite regularly in the picture, and there are likely many more switches not visible because of *Performance Monitors* low sampling resolution.

Scalability

The *Cpuhog* example above demonstrates how the flaws in the NT scheduling algorithm can affect a partially loaded, relatively static machine. One can easily envision scenarios where a 4 or 8 processor server is running several enterprise-level server applications, each with a dozen or more threads spread across the priority spectrum. Low priority threads handle background work, mid-range threads update graphical monitors, process user input and handle connections, and real-time threads process time-critical operations.

Under medium to heavy loads Steps 2 and 3 of **KiReadyThread** could cause a significant number of needless thread migrations per second, where high priority threads would constantly displace lower priority threads that would be forced to migrate. The overhead of these switches would then be a measurable factor on system performance, and lead to server applications not scaling well as processors are added.

What's the Fix?

What's surprising about the NT scheduler flaw of Step 2 of **KiReadyThread** and the unnecessary switches of Step 3, is not so much the fact that they have been there for several years, but that they were designed into NT in the first place. A common-sense way to perform priority-preserving scheduling across multiple processors is to check the lowest priority executing threads. The algorithm would start with the lowest priority thread that is running on a CPU the ready thread has hard affinity for, and see if the ready thread is higher priority. If its priority is higher then it will preempt the lower priority thread. If it has a lower priority it will be placed in the ready list for its priority. This algorithm preserves priority globally across the processors and thus prevents wasted thread migrations. Sun's *Solaris* operating system employs this algorithm.

An SMP scheduling algorithm could attempt to maintain soft-affinity, but doing so has complex trade-offs that are certainly not reflected in the NT algorithm. For one thing, NT's use of soft-affinity ignores elapsed time. Second, it uses a 20ms rule in **KiFindReadyThread** that is a hard-coded value. Both are independent of the factors that determine reasonable time windows, like processor speed and cache size.
