Writing Scalable Applications for Windows NT

John Vert
Windows NT Base Group

Revision 1.0: June 6, 1995

1. Introduction

One of the major goals of Microsoft® Windows NT™ is to provide a robust, scalable symmetric multiprocessing (SMP) operating system. By harnessing the power of multiple processors, SMP can deliver a large boost to the performance and capacity of both workstation and server applications. Scalable SMP client-server systems can be very effective platforms for downsizing traditional mainframe applications. SMP also makes it easy to increase the computing capacity of a heavily loaded server by adding more processors. Windows NT provides a great platform to base server applications on, but the operating system is not solely responsible for performance and scalability. Applications hosted on Windows NT must also be designed with these goals in mind. A perfectly efficient system devotes all of its resources to any given problem. But without a carefully designed application, unnecessary system code will be executed, leaving fewer resources available for the application to devote to the problem.

Windows NT provides many advanced features to make the development of efficient, scalable applications easier. Other SMP operating systems have some of these features, but a few are unique to Windows NT. Understanding and using these features is the key to realizing the full potential of Windows NT SMP in your application. This article will cover the use of these features, and describe some of the common pitfalls encountered in SMP programming. It assumes you have a good working knowledge of Win32®, particularly overlapped input/output (I/O) and network programming.

2. Threads

Windows NT's basic unit of execution is the *thread*. Each process contains one or more threads. Threads can run on any processor in a multiprocessor system, so splitting a single-threaded program into multiple concurrent threads is a quick way to take advantage of SMP systems. A similar approach splits a single-threaded server into multiple server processes. Traditional UNIX® SMP operating systems that do not have native support for threads often use this approach. Since processes require more system overhead than threads, a single multithreaded program is a more efficient solution on Windows NT.

In order to effectively split a single-threaded program into multiple threads, you need to understand how threads work. Normally, a thread can be in one of three states at a given time.

- **Waiting**—The thread cannot run until a specified event occurs.
- **Ready**—The thread is ready to run, but no processor is currently available.
- **Running**—The thread is currently running on a processor.

Any thread in either the ready or running state is runnable and may profitably use any available control processing unit (CPU) cycles. The number of runnable threads is limited only by system resources, but the number of currently running threads is limited by the number of processors in the system.

2.1 The Windows NT Scheduler

The Windows NT kernel is responsible for allocating the available CPUs among the system's runnable threads in the most efficient manner. To do this, Windows NT uses a priority-based round-robin algorithm. The Windows NT kernel supports 31 different priorities, and a queue for each of the 31 priorities contains all the ready threads at that priority. When a CPU becomes available, the kernel finds the highest priority queue with ready threads on it, removes the thread at the head of the queue, and runs it. This process is called a *context switch*.

The most common reason for a context switch is when a running thread needs to wait. This happens for a number of different reasons. If a thread touches a page that is not in its working set, the thread must wait for memory management to resolve the page fault before it can continue. Many system calls, such as **WaitForSingleObject** or **ReadFile**, explicitly block the running thread until the specified event occurs.

When a running thread needs to wait, the kernel picks the highest-priority ready thread and switches it from the ready state to the running state. This ensures that the highest priority runnable threads are always running. To prevent CPU-bound threads from monopolizing the processor, the kernel imposes a time limit (called the *thread quantum*) on each thread. When a thread has been running for one quantum, the kernel preempts it and moves it to the end of the ready queue for its priority. The actual length of a thread's quantum currently varies from 15 milliseconds to 30 milliseconds across different Windows NT platforms, but this may change in future versions.

Another reason for a context switch is when an event changes a higher-priority thread's state from waiting to ready. In this case, the higher-priority thread will immediately preempt a lower-priority thread running on the processor.

On a uniprocessor computer, only one thread can be in the running state, because there is only one processor. A multiprocessor computer allows for one running thread per processor. It is important to understand that a multiprocessor computer will *not* make a single thread complete its activity any faster. The entire performance gain is a result of multiple threads running simultaneously. Even if your application uses multiple threads, the threads must be able to work independently of each other to scale effectively. If your application is too serialized (meaning that threads have interdependencies that force them to wait for each other) there will not be enough runnable threads to distribute across the processors, and some processors may be idle. Adding more processors will only increase the total CPU time spent idle— it will not make your application run much faster.

Another factor to consider is whether your application is *compute-bound* (limited by the speed of the CPU) or *I/O-bound* (limited by the speed of some I/O device, typically disk drivers or network bandwidth). If your application is I/O bound and cannot saturate the CPU of a uniprocessor computer, adding more processors is unlikely to make it run much faster. If your application spends most of its time waiting for the disk, additional CPUs will just tighten the I/O bottleneck.

2.1 How Many Threads Do I Need?

There are two basic models for implementing a client-server application. The easiest to use is a single thread that services all client requests in turn. However, this model suffers from the "too few threads" syndrome, and will not work any faster on an SMP computer. It's not even a very good model for a uniprocessor computer, because if the single thread ever blocks, no application work can be done. Blocking on I/O, for example, is almost inevitable in any application. This model can be stretched to one thread per processor, but the problem of evenly dividing client requests among the threads is difficult to solve efficiently. If one of the threads needs to wait for I/O, there is still no way to prevent its CPU from idling until the wait completes.

The model at the other end of the spectrum creates one thread for each client. This readily solves the problem of providing enough threads to utilize all the CPUs. Because each client has its own dedicated thread, there is no need to manually balance the entire client load over a few threads. But this is an expensive solution that degrades with large numbers of clients. As the number of ready threads becomes much greater than the number of processors, overall performance decreases. Each thread spends more time waiting on the ready queue for its turn to run, and the kernel must spend more time context switching the threads in and out of the running state.

Threads are not free, so a design that uses hundreds of ready threads can consume quite a lot of system resources in the form of memory and increased scheduling overhead. Windows NT can swap out the memory resources used by a waiting thread, but before a thread can become runnable, its resources must be brought back into memory.

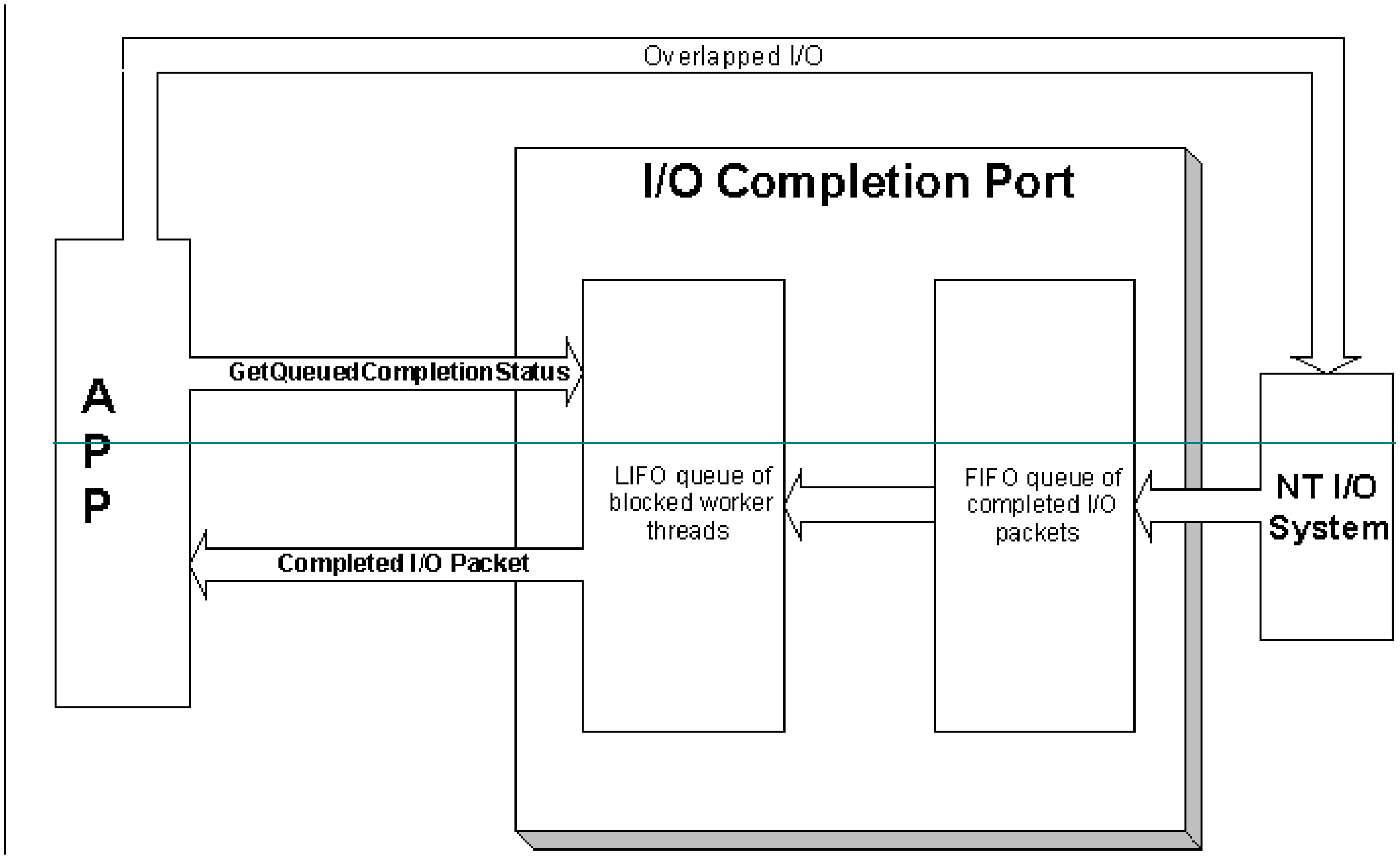
3. I/O Completion Ports

An ideal model would strike a balance between the two extremes. There should always be enough runnable threads to fully utilize the available CPUs, but there should never be so many threads that the overhead becomes too large. In fact, the ideal number of runnable threads is not related to the number of clients at all, but to the number of CPUs in the server. Unfortunately, multiplexing a large number of clients across a smaller number of runnable threads is difficult for an application to do. The application cannot always know when a given thread is going to block, and without this knowledge it cannot activate another thread to take its place. To solve this problem and make it easy for programmers to write efficient, scalable applications, Windows NT version 3.5 provides a new mechanism called the *I/O completion port*.

An I/O completion port is designed for use with overlapped I/O. **CreateIoCompletionPort** associates the port with multiple file handles. When asynchronous I/O initiated on any of these file handles completes, an I/O completion packet is queued to the port. This combines the synchronization point for multiple file handles into a single object. If each file handle represents a connection to a client (usually through a named pipe or socket), then a handful of threads can manage I/O for any number of clients by waiting on the I/O completion port. Rather than directly waiting for overlapped I/O to complete, these threads use **GetQueuedCompletionStatus** to wait on the I/O completion port. Any thread that waits on a completion port becomes associated with that port. The Windows NT kernel keeps track of the threads associated with an I/O completion port.

Of course, **WaitForMultipleObjects** can produce similar behavior, so there must be a better reason for inventing I/O completion ports. Their most important property is the controllable concurrency they provide. An I/O completion port's concurrency value is specified when it is created. This value limits the number of runnable threads associated with the port. When a thread waits on a completion port, the kernel associates it with that port. The kernel tries to prevent the total number of runnable threads associated with a completion port from exceeding the port's concurrency value. It does this by blocking threads waiting on an I/O completion port until the total number of runnable threads associated with the port drops below its concurrency value. As a result, when a thread calls **GetQueuedCompletionStatus**, it only returns when completed I/O is available, and the number of runnable threads associated with the completion port is less than the port's concurrency. The kernel dynamically tracks the completion port's runnable threads. When one of these threads blocks, the kernel checks to see if it can awaken a thread waiting on the completion port to take its place. This throttling effectively prevents the system from becoming swamped with too many runnable threads. Because there is one central synchronization point for all the I/O, a small pool of worker threads can service many clients.

Unlike the other Win32 synchronization objects, threads that block on an I/O completion port (by using **GetQueuedCompletionStatus**) unblock in last in, first out (LIFO) order. Because it does not matter which thread services an I/O completion, it makes good sense to wake the most recently active thread. Threads at the bottom of the stack have been waiting for a long time, and will usually continue to wait, allowing the system to swap most of their threads' memory resources out to disk. Threads near the top of the stack are more likely to have run recently, so their memory resources will not be swapped to disk or flushed from a processor's cache. The net result is that the number of threads waiting on the I/O completion port is not very important. If more threads block on the port than are needed, the unused threads simply remain blocked. The system will be able to reclaim most of their resources, but the threads will remain available if there are enough outstanding transactions to require their use. A dozen threads can easily service a large set of clients, although this will vary depending on how often each transaction needs to wait. Note that the LIFO policy only applies to threads that block on the I/O completion port. The completion port delivers completed I/O in first in, first out (FIFO) order. See the figure below.


Overlapped I/O
I/O Completion Port
APP
GetQueuedCompletionStatus
Completed I/O Packet
LIFO queue of blocked worker threads
FIFO queue of completed I/O packets
NT I/O System

Tuning the I/O completion port's concurrency is a little more complicated. The best value to pick is usually one thread per CPU. This is the default if zero is specified at the creation of the I/O completion port. There are a few cases where a larger concurrency is desirable. For example, if your transaction requires a lengthy computation that will rarely block, a larger concurrency value will allow more threads to run. The kernel will preemptively timeslice among the running threads, so each transaction will take longer to complete. However, more transactions will be processing at the same time, rather than sitting in the I/O completion port's queue, waiting for a running thread to complete. Simultaneously processing more transactions allows your application to have more concurrent outstanding I/O, resulting in higher use of the available I/O throughput. It is easy to experiment with different values for the I/O completion port's concurrency and see their effect on your application.

The standard way to use I/O completion ports in a server application is to create one handle for each client [by using **ConnectNamedPipe** or **listen**, depending on the interprocess communication (IPC) mechanism], and then call **CreateIoCompletionPort** once for each handle. The first call to **CreateIoCompletionPort** will create the port. Subsequent calls associate additional handles with the port. After a client establishes a connection and the handles are associated with the I/O completion port, the server application posts an overlapped read to the client's handle. When the client writes a request to the server, this read completes and the I/O system queues an I/O completion packet to the completion port. If the current number of runnable threads for the port is less than the port's concurrency, the port will become signaled. If there are threads waiting on the port, the kernel will wake up the last thread (remember, waits on I/O completion ports are satisfied in LIFO order) and hand it the I/O completion packet. When there are no threads currently waiting on the port, the packet is handed to the next thread that calls **GetQueuedCompletionStatus**. The Windows NT 3.5 Software Development Kit contains source code for SOCKSRV, a simple network server that demonstrates this technique.

The most efficient scenario occurs when there are I/O completion packets waiting in the queue, but no waits can be satisfied because the port has reached its concurrency limit. In this case, when a running thread completes a transaction, it calls **GetQueuedCompletionStatus** to pick up its next transaction and immediately picks up the queued I/O packet. The running thread never blocks, the blocked threads never run, and no context switches occur. This demonstrates one of the most interesting properties of I/O completion ports—the heavier the load on the system, the more efficient they are. In the ideal case, the worker threads never block, and I/O completes to the queue at the same rate that threads remove it. There is always work on the queue, but no context switches ever need to occur. After a thread completes one transaction, it simply picks the next one off the completion port and keeps going.

Occasionally, an application thread may need to issue a synchronous read or write to a handle associated with an I/O completion port. For example, a network server may get partially through one transaction before discovering it needs more data from the client. A normal read would signal the I/O completion port, causing a different thread to pick up the I/O completion and process the remainder of the transaction. For this reason, Win32 extends the semantics of **ReadFile** and **WriteFile** to allow an application to override the I/O completion port mechanism on a per-I/O basis. The application makes a normal overlapped call to **ReadFile** or **WriteFile** with an **OVERLAPPED** structure that contains a valid hEvent handle. To distinguish this call from the normal case, the application also sets the low bit of the hEvent handle. Because Win32 reserves the low two bits of a handle, **ReadFile** and **WriteFile** use the low bit as a "magic bit" to indicate that this particular I/O should not complete to the I/O completion port. Instead, the application uses the normal Win32 overlapped completion mechanism (wait on hEvent, or call **GetOverlappedResult** with fWait==TRUE).

Windows NT 3.51 adds one more application programming interface (API) for dealing with I/O completion ports. **PostQueuedCompletionStatus** lets an application queue its own special-purpose packets to the I/O completion port without issuing any I/O requests. This is useful for notifying worker threads of external events. For example, a clean shutdown might require each thread waiting on the I/O completion port to perform thread-specific cleanup before calling **ExitThread**. Posting one application-defined "cleanup and shutdown" packet for each worker thread notifies each worker thread to clean up and exit. Because the caller of **PostQueuedCompletionStatus** has complete control over all the return values of **GetQueuedCompletionStatus**, an application can define its own protocol for recognizing and handling these packets.

4. Synchronization Primitives

Win32 provides a wide assortment of synchronization objects. These objects include events (both auto-reset and manual-reset), mutexes, semaphores, critical sections, and even raw interlocked operations. For simply protecting access to a data structure, all these objects will work fine. But in some circumstances, tthey can vary dramatically in performance. Any efficient server application must understand the tradeoffs inherent in each synchronization object.

4.1 Mutexes, Events, and Semaphores

Mutexes, events, and semaphores are all powerful synchronization objects provided directly by the Windows NT kernel. Because they are real Win32 objects, they are inheritable, have security descriptors, and can be named and used to synchronize multiple processes. **WaitForMultipleObjects** (and **MsgWaitForMultipleObjects**) provide plenty of power and flexibility when combining multiple synchronization objects of this type.

The kernel directly manages the synchronization of these objects, and will perform an immediate context switch when a thread blocks on one of these objects. Because accessing synchronization objects requires a kernel call, some overhead is involved. In cases where the synchronization period is very short and very frequent, this overhead (and any resulting context switches) may be much greater than the synchronization period.

4.2 Critical Sections

Unlike the flexible kernel synchronization objects, a Win32 critical section does only one thing. A critical section is a very fast method for mutual-exclusion within a single multithreaded process. **EnterCriticalSection** grants exclusive ownership of a critical section object, and **LeaveCriticalSection** releases it. Because critical sections are not handle-based objects, they cannot be named, secured, or shared across multiple processes. They also cannot be used with **WaitForMultipleObjects** or even **WaitForSingleObject**. This simplicity allows them to be very fast at mutual-exclusion. When there is no contention for a critical section, only a few instructions are needed to acquire or release it. When contention for a critical section occurs, a kernel synchronization object is automatically used to allow threads to wait for and release the critical section. As a result, critical sections are usually the fastest mechanism for protecting critical code or data. The critical section only calls the kernel to context switch when there is contention and a thread must either wait or awaken a waiting thread.

4.3 Spinlocks

In rare cases, it may be necessary to build your own synchronization mechanism. On an SMP system, normal memory references are not atomic. If two processors are simultaneously modifying the same memory location, one of the processor's updates will be lost. Performing atomic memory updates requires special processor instructions. X86 architectures provide the LOCK prefix to exclusively lock the memory bus for the duration of an instruction. RISC architectures (such as Mips, Alpha and PowerPC) provide a load-linked/store-conditional sequence of instructions for atomic updates. There are three Win32 APIs—**InterlockedIncrement**, **InterlockedDecrement**, and **InterlockedExchange—**that use these instructions to perform atomic memory references in a portable fashion. They can be used to implement spinlocks or reference counts without relying on the Win32 synchronization primitives. Do not confuse application-level spinlocks with the kernel spinlocks used internally by the Windows NT executive and I/O drivers.

The performance of the interlocked routines varies greatly, depending on the underlying hardware. Processor architecture, memory bus design, and cache effects all have a big impact on how fast the hardware can perform interlocked operations. One way to implement a spinlock is to use a value of zero to represent a free spinlock. When a thread needs to acquire the spinlock, it uses **InterlockedExchange** to set its value to 1. The spinlock is acquired if the result of the **InterlockedExchange** is 0, otherwise the attempt has failed and must be retried. There are many different strategies for retrying the lock acquisition (or, "spinning"). The best method depends on many factors. The hardware, memory cache policy, frequency of lock acquisition, and length of time the lock is held all make a difference. An important point to remember when selecting a retry policy is that the interlocked routines can be very expensive in their use of the system's memory bus. Spinning in a loop of **InterlockedExchange** calls is a good way to reduce the available memory bandwidth and slow down the rest of the system. It is better to read the lock's value in a loop, and only retry the **InterlockedExchange** when the lock appears free.

Spinlocks are a very efficient method of synchronizing small sections of code, but they do have some serious drawbacks. If the thread that owns the spinlock blocks for any reason, (to wait for I/O or a page fault, for example) all the threads in the application system must spin on the lock until the owning thread completes its wait and releases the lock. Because the Windows NT kernel cannot tell the difference between spinning on a spinlock and doing useful work, threads that are doing nothing but spinning will waste valuable CPU time. Even if the thread does not block, the kernel may summarily preempt it when it uses up its quantum or a higher priority thread becomes runnable. Again, because the Windows NT kernel does not know when a thread owns a spinlock, there is no way it can avoid preempting threads before they have a chance to release the lock. A Win32 synchronization object, on the other hand, immediately context switches to another runnable thread instead of wasting time spinning.

Boosting a thread's priority to real time will protect it from being preempted by most of the other threads in the system, but this is a fairly drastic solution. If real-time threads use all the CPUs, the system will appear completely frozen. This makes it hard to debug your application or even terminate it if something goes wrong! Because of these risks, Windows NT controls access to real-time threads that use its security model. Only processes running in a user account with the right to increase scheduling priority may change their thread priority to one of the potentially dangerous levels. Even real-time priority is not a complete solution, because the kernel scheduler round-robins real-time threads at the same priority. If a thread acquires a spinlock, then exhausts its quantum, any other thread trying to acquire the lock must spin until the owning thread is rescheduled and releases the spinlock. To solve this, sophisticated algorithms for spinning can be developed that detect when the owning thread has spent too much time spinning and should yield to another thread. A thread can yield the remainder of its quantum by calling **Sleep** with a sleep time of zero milliseconds. When this occurs, a yielding thread goes to the end of its ready queue, and another thread of the same priority can be given the CPU. A backoff spin algorithm like this is difficult to tune correctly for complex applications. Determining how long a thread should spin before giving up and yielding is critically important. Unnecessary yielding negates any performance advantage spinlocks have over critical sections, while insufficient yielding can occasionally cause drastic performance degradation if a thread is preempted while owning a spinlock.

A spinlock's limitations are severe enough that they should not be considered unless your application cannot tolerate the overhead of blocking in the kernel when a lock is owned. For this reason, you should use critical sections instead.

5. Managing Memory Usage

Processor cycles are not the only resource managed by the operating system. Efficient use of physical memory is critical to performance. Windows NT balances the available physical memory between the system, the file cache, and the applications by using working sets. The working set of a process consists of the set of resident physical pages visible to the process. When a thread accesses a page that is not in the working set of its process, a page fault occurs. Before the thread can continue, the virtual memory manager must add the page to the working set of the process. A larger working set increases the probability that a page will be resident in memory, and decreases the rate of page faults. One of the most critical parameters for a file server is the size of the file cache's working set. So in order to maximize file server performance, a default Windows NT Server installation increases the file cache at the expense of applications.

On the other hand, an application server's performance depends more on the working set of the application than on the size of the file cache, so this parameter should be changed on an application server. To change this setting, choose the Network applet from Control Panel. In the Installed Network Software box, click Server, then click Configure. This brings up a dialog box that lets you change this parameter to favor application performance.

Windows NT tries to do a good job of sharing physical memory between the system and the application, but sometimes an application needs to reserve more memory resources than it would normally get. To allow applications to request this special treatment, Windows NT version 3.5 introduces two new Win32 APIs: **GetProcessWorkingSetSize** and **SetProcessWorkingSetSize**. As with real-time priority threads, increasing an application's working set requires great care in order to avoid unpleasant effects on the rest of the system. Also like real-time priority threads, the process must hold the privilege to increase scheduling priority in order to override the system's decisions with these APIs. Although the system will do its best to honor the working set limits, low-memory situations can cause a process's working set to drop below the minimum size. If an application needs to force certain pages of memory to remain resident, it must use **VirtualLock**.

6. Caches

Computers that run Windows NT generally have a fast memory cache between the CPU and main memory. This takes advantage of memory access locality to allow most of the CPU's memory references to complete at the speed of the fast cache memory, instead of the much slower speed of main memory. Without this cache, the slower speed of DRAM memory would cripple the performance of modern, high-speed processors. In SMP systems, cache memory has an additional function that is vital to system performance. Each processor's memory cache also insulates the main shared memory bus from the full memory bandwidth demand of the combined processors. Any memory access that the cache can satisfy will not need to burden the shared memory bus. This leaves more bandwidth available for the other processors.

Any system that uses caches depends on the locality of memory accesses for good performance. SMP systems that provide separate caches for each processor introduce additional issues that affect application performance. Memory caches must maintain a consistent view of memory for all processors. This is accomplished by dividing memory into small chunks (that make up a *cache line*) and by tracking the state of each chunk present in one of the caches. To update a cache line, a processor must first gain exclusive access to it by invalidating all other copies in other processors' caches. When the processor has exclusive access to the cache line, it may safely update it. If the same cache line is continuously updated from many different processors, that cache line will bounce from one processor's cache to another. Because the processor cannot complete the write instruction until its cache acquires exclusive access to the cache line, it must stall. This behavior is called *cache sloshing*, because the cache line "sloshes" from one processor's cache to another.

One common cause of cache sloshing is when multiple threads continuously update global counters. You can easily fix these counters by keeping separate variables for each thread, then summoning them when required. A more subtle variant of the problem occurs when two or more variables occupy the same cache line. Updating any of the variables requires exclusive ownership of the cache line. Two processors updating different variables will slosh the cache line as much as if they were updating the same variable. You can be remedy this by simply padding data structures to ensure that frequently accessed variables do not share a cache line with anything else. Packing variables that are frequently accessed together into a single cache line can also improve performance by reducing the traffic on the memory bus. Most current systems have 32-byte cache lines, although cache lines of 64 bytes or more will show up in future systems.

Cache sloshing can be simple to fix, but very difficult to find. A profiling tool that tracks the total time spent in different functions is helpful, but plenty of guesswork and intuition is still necessary. Compare profiles of your program running on configurations with different numbers of processors. Any functions that take proportionally more time as the number of processors increases are likely victims of cache sloshing. As more processors compete for the same cache lines, the instructions that access those cache lines will run slower and slower. The function will not actually execute more instructions, but each instruction that needs to wait for the cache will take longer to complete, thus increasing the total time spent in the function.

7. Conclusion

After carefully designing and tuning your application, you may find that it still does not scale well. Sometimes, it is not only the software that is responsible. The availability of Windows NT has inspired an explosion of SMP computer designs. These computers range across a broad spectrum from personal workstations to million-dollar superservers. Building an SMP system presents even more design tradeoffs than building a uniprocessor system. As a result, many SMP computers vary widely in their overall performance and scalability. Because no industry standard benchmark for these computers has emerged yet, comparisons of different platforms can be difficult. Of course, existing benchmarks can test common components such as the disk or video subsystem. However, the memory bus, one of the most critical performance parameters for an SMP computer, can easily become a bottleneck for SMP applications.

While most computers scale well when the application runs mainly in the memory cache, many applications require working sets that are much larger than the cache. When multiple processors are continually contending for access to the main memory bus, the total main memory bandwidth is very important. The listing below contains the source code for MEMBENCH, a short program that tests the raw memory throughput of SMP computers. MEMBENCH measures the time required for multiple threads to modify a large array. When each thread accesses the array sequentially, locality is high and scalability is very good. As the stride used to step through memory increases, the locality decreases, causing more cache misses and dramatically decreasing the overall throughput and scalability.

```
--*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct _THREADPARAMS {
    DWORD ThreadIndex;
    PCHAR BufferStart;
    ULONG BufferLength;
    DWORD Stride;
} THREADPARAMS, *PTHREADPARAMS;


DWORD MemorySize = 64*1024*1024;
HANDLE StartEvent;
THREADPARAMS ThreadParams[32];
HANDLE ThreadHandle[32];
ULONG TotalIterations = 1;


DWORD WINAPI
MemoryTest(
    IN LPVOID lpThreadParameter
    );

main (argc, argv)
    int argc;
    char *argv[];
{
    DWORD CurrentRun;
    DWORD i;
    SYSTEM_INFO SystemInfo;
    PCHAR Memory;
    PCHAR ThreadMemory;
    DWORD ChunkSize;
    DWORD ThreadId;
    DWORD StartTime, EndTime;
    DWORD ThisTime, LastTime;
    DWORD IdealTime;
    LONG IdealImprovement;
    LONG ActualImprovement;
    DWORD StrideValues[] = {4, 16, 32, 4096, 8192, 0};
    LPDWORD Stride = StrideValues;
    BOOL Result;

    //
    // If you have an argument, use that as the number of iterations.
    //
    if (argc > 1) {
        TotalIterations = atoi(argv[1]);
        if (TotalIterations == 0) {
            fprintf(stderr, "Usage: %s [# iterations]\n",argv[0]);
            exit(1);
            }
            printf("%d iterations\n",TotalIterations);
        }
        //
        // Determine how many processors are in the system.
        //
        GetSystemInfo(&SystemInfo);

        //
        // Create the start event.
        //
        StartEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
        if (StartEvent == NULL) {
            fprintf(stderr, "CreateEvent failed, error %d\n",GetLastError());
            exit(1);
        }

        //
        // Try to boost your working set size.
        //
        do {
            Result = SetProcessWorkingSetSize(GetCurrentProcess(), MemorySize,
                     MemorySize*2);
            if (!Result) {
                MemorySize -= 10*1024*1024;
            }
        } while ( !Result );

        printf("MEMBNCH: Using %d MB array\n", MemorySize / (1024*1024));

        //
        // Allocate a big chunk of memory (64MB).
        //
        Memory = VirtualAlloc(NULL,
                              MemorySize,
                              MEM_COMMIT,
                              PAGE_READWRITE);
        if (Memory==NULL) {
            fprintf(stderr, "VirtualAlloc failed, error %d\n",GetLastError());
            exit(1);
        }

        do {
            printf("STRIDE = %d\n", *Stride);
            for (CurrentRun=1; CurrentRun<=SystemInfo.dwNumberOfProcessors;
                 CurrentRun++) {

                printf("  %d threads: ", CurrentRun);
                //
                // Start the threads, and let them party on the
                // memory buffer.
                //
                ResetEvent(StartEvent);
        ChunkSize = (MemorySize / CurrentRun) & ~7;

        for (i=0; i<CurrentRun; i++) {
            ThreadParams[i].ThreadIndex = i;
            ThreadParams[i].BufferStart = Memory + (i * ChunkSize);
            ThreadParams[i].BufferLength = ChunkSize;
            ThreadParams[i].Stride = *Stride;

            ThreadHandle[i] = CreateThread(NULL,
                                           0,
                                           MemoryTest,
                                           &ThreadParams[i],
                                           0,
                                           &ThreadId);
            if (ThreadHandle[i] == NULL) {
                fprintf(stderr, "CreateThread %d failed, %d\n", i,
                        GetLastError());
                exit(1);
            }
        }

        //
        // Touch all the pages.
        //
        ZeroMemory(Memory, MemorySize);

        //
        // Start the threads and wait for them to exit.
        //
        StartTime = GetTickCount();
        SetEvent(StartEvent);

        WaitForMultipleObjects(CurrentRun, ThreadHandle, TRUE, INFINITE);
        EndTime = GetTickCount();

        ThisTime = EndTime-StartTime;

        printf("%7d ms",ThisTime);
        printf(" %.3f MB/sec",(float)(MemorySize*TotalIterations)/
               (1024*1024) / ((float)ThisTime / 1000));

        if (CurrentRun > 1) {
            IdealTime = (LastTime * (CurrentRun-1)) / CurrentRun;
            IdealImprovement = LastTime - IdealTime;
            ActualImprovement = LastTime - ThisTime;
            printf("  (%3d %% )\n",(100*ActualImprovement)/
                   IdealImprovement);
        } else {
            printf("\n");
        }
        LastTime = ThisTime;
```

```
            for (i=0; i<CurrentRun; i++) {
                CloseHandle(ThreadHandle[i]);
            }
        }

        ++Stride;
    } while ( *Stride );
}

DWORD WINAPI
MemoryTest(
    IN LPVOID lpThreadParameter
    )
{
    PTHREADPARAMS Params = (PTHREADPARAMS)lpThreadParameter;
    ULONG i;
    ULONG j;
    DWORD *Buffer;
    ULONG Stride;
    ULONG Length;
    ULONG Iterations;

    Buffer = (DWORD *)Params->BufferStart;
    Stride = Params->Stride / sizeof(DWORD);
    Length = Params->BufferLength / sizeof(DWORD);
    WaitForSingleObject(StartEvent,INFINITE);

    for (Iterations=0; Iterations < TotalIterations; Iterations++) {
        for (j=0; j < Stride; j++) {

            for (i=0; i < Length-Stride; i += Stride) {

                Params->BufferStart[i+j] += 1;
            }
        }
    }
}
```

As with any application that needs tuning, developing a scalable application requires attention to small details and careful design. Windows NT provides an excellent framework for taking advantage of powerful SMP platforms, but it also provides features that are unfamiliar to most programmers. Understanding how to combine powerful features such as threads, asynchronous I/O, and completion ports is the key to unlocking the performance that Windows NT offers.