NetWare 6 has been on the horizon long enough that you probably know at least this much: NetWare 6 is designed to run on symmetric multiprocessing (SMP) servers. SMP servers offer the power of as few as two, as many as 32, but most commonly (today, at least) four CPUs (hereafter called processors). Any of these processors can process any task, and all of them can process tasks simultaneously.

One of the potential rewards of using SMP servers is that you can increase processing power without adding servers. (For more information about the increase in processing power you can expect for each additional processor on an SMP server, see "Up to Scale" on p. 8.) SMP hardware not only enables you to increase processing power on a single box but also indirectly increases security and improves manageability: After all, protecting and managing one server with eight processors is easier than protecting and managing eight separate servers.

Of course, to reap these potential rewards, you’ll need at the very least a multiprocessing (MP)-enabled operating system such as NetWare 6. If you’re at all curious about the way things work, the last sentence may leave you asking, What does it really mean to say an operating system is “MP-enabled”? Naturally, saying that NetWare 6 is MP-enabled suggests that it can run on multiple processors, but how? And, exactly which components in NetWare 6 are MP-enabled anyway?

Just for the record, you don’t have to know how NetWare 6 works to use it. Of course, you don’t have to know how a toaster works either. However, since you were probably the type of kid who took apart the toaster for fun—just to see how it worked—you’re probably interested in lifting the NetWare 6 hood. To that end, we’re offering you this bare bones, quasi-technical explanation to give you an idea of how NetWare 6 works.

A multithreaded program that is not MP-enabled consists of two or more threads—that is, sequences of executing code—that a single processor can execute concurrently. Despite what concurrently connotes outside of this industry, saying that multiple threads can execute concurrently is not the same as saying that they can execute simultaneously. Multithreaded programs that are not MP-enabled are designed to run on only one processor and will run on only one processor no matter how many processors are available.

One processor can execute only one thread at a time. Concurrency suggests that the processor can switch between multiple threads so efficiently that users may feel as if the processor is executing threads simultaneously. What is really happening is that while one thread is executing, other threads are in a state of suspension. The processor can switch from one thread to another at any point—usually at the point the currently executing thread completes or relinquishes the processor.

For example, Thread Red may have to wait for a certain event (for example, the setting of an integer) before it can continue processing. While Thread Red waits, it removes itself from the processor’s run queue. The processor is then free to execute other threads (Thread Blue and Thread Green, for example) while Thread Red waits for the event. After the event occurs, Thread Red returns to the processor’s run queue, and the processor dutifully processes the thread.

Illusion Versus Reality

Multithreaded applications enable multitasking. A multithreaded operating system that supports multitasking can execute threads from different multithreaded programs concurrently on a single processor. Because a multitasking operating system can juggle multiple programs, it creates for users the illusion that a single-processor computer is executing multiple programs simultaneously. NetWare versions prior to NetWare 5 were not MP-enabled, with the exception of NetWare 4.
SPREADING THREADS

When you install NetWare 6, how does it spread its own and other programs’ threads? For starters, NetWare 6 utilizes a multiprocessor kernel (MPK) to determine how many processors it has to work with. (A kernel, as you probably know, is the essential part of an operating system. For example, the kernel is responsible for resource allocation and hardware interfaces.) NetWare 6 uses the MP 1.4 Platform Support Module (MP14.PSM) to detect the number of processors available upon installation.

After the NetWare 6 MPK knows how many processors are available, it must decide where to send threads as they present themselves for the first time. The portion of the NetWare 6 MPK that makes this determination and thereafter coordinates thread processing is appropriately called the Scheduler. The Scheduler decides how to distribute threads based on information about either the threads or the processors.

For example, a program developer may flag a program as MP-safe, which means that the program is non-MP-enabled but safe to run in an MP environment. The Scheduler places MP-safe threads on Processor 0, which processes all non-MP-enabled threads. Novell engineer Bruce Rogers points out what this fact implies: that NetWare 6 can safely run “correctly written” applications that were developed before MP became an issue. “These applications,” notes Rogers, “do not need to be modified in any way. NetWare 6 provides a scheduling environment (on Processor 0) that isolates these applications from the surrounding MP environment.”

Programs may also indicate that they want to bind to specific processors. In these cases, the Scheduler sends the thread to the processor for which the program expresses a preference. However, Novell engineer Dana Henricksen is quick to point out that Novell strongly discourages this practice. Nonetheless, Henricksen adds, Novell has made allowances for thread-to-processor bindings because management utilities and a few other programs need to execute certain threads on specific processors.

If a thread is MP-enabled and its parent program has not indicated that the thread needs to run on a specific processor, then the Scheduler checks to see if any processors are idle. For example, suppose you install NetWare 6 on a brand-new four-processor SMP server. As these generic threads present themselves, the Scheduler checks the processors and finds they are all idle. In this case, the Scheduler sends the first thread to Processor 0, the second thread to Processor 1, the third thread to Processor 2, and so on until all of the processors are busy.

 Threads—Let Them Be

After placing a thread on a processor, the Scheduler generally leaves the thread there. In fact, the Scheduler moves a thread’s execution from one processor to another only when absolutely necessary. The Scheduler moves a thread from one processor to another under the following two circumstances:

• The thread is from a program that is not MP-enabled, in which case the Scheduler moves the thread to the safety of the legacy scheduling environment on Processor 0, a process called funneling.

Up to Scale

Symmetric multiprocessing (SMP) and were particularly good at multitasking. However, make no mistake: A multithreaded, multitasking operating system running on only one processor cannot execute more than one thread at a time.

Even multiple processors cannot simultaneously execute threads from multithreaded programs unless those programs are MP-enabled. MP-enabled programs are written in such a way that their threads can safely execute simultaneously on multiple processors. Hence, an MP-enabled operating system such as NetWare 6 changes the illusion of completing tasks simultaneously into reality.

The Right Hardware Stuff

You should run the MP-enabled NetWare 6 on SMP servers that support Intel’s MultiProcessor Specification (MPS) 1.4. Fortunately, these servers are not hard to find: All Intel-based SMP hardware vendors, including Compaq Computer Corp. and Dell Computer Corp., support MPS 1.4.

MPS 1.4 defines a model for SMP hardware in which all processors are functionally identical, have equal status, and can communicate with one another. Furthermore, all of the processors in SMP hardware (hardware that complies with MPS 1.4) share the same I/O subsystem and also the same memory space, which they access using the same memory addresses. As a result, all of the processors can execute one copy of an MP-enabled operating system such as NetWare 6. (For more information about MPS 1.4, you can download the densely technical documentation specification at http://developer.intel.com/design/interch/MANUALS/242016.htm.)

- The thread is from a program that is not MP-enabled, in which case the Scheduler moves the thread to the safety of the legacy scheduling environment on Processor 0, a process called funneling.
Spread the Threads: MP-Enabled Components

In NetWare 6, many of the core components and services upon which the NetWare kernel depends will be MP-enabled. These MP-enabled components will include the following:

**PROTOCOL STACKS**
- IP stack
- HTTP
- Web-based Distributed Authoring and Versioning (WebDAV)
- Lightweight Directory Access Protocol (LDAP)
- NetWare News Server
- NetWare Core Protocol (NCP)
- Service Location Protocol (SLP) 2
- Gigabit Ethernet, 100 Megabit Ethernet, 10 Megabit Ethernet
- Token Ring 16

- The load-balancing mechanism detects a gross imbalance among the processors, in which case NetWare 6 may migrate one or more threads to evenly distribute the workload among available processors.

The Scheduler's load-balancing mechanism is conservative, moving threads from one processor to another only when the thread load on one processor is significantly higher than the average. When you use NetWare 6 and can't suppress your curious nature, you can use the NetWare management portal to see how many threads were moved for load-balancing purposes within a given time frame. To do this, look for the Threads Moved to Other CPU field under Kernel Statistics in the System Statistics option.

A side from funneling non-MP-enabled threads and moving threads for load-balancing purposes, the Scheduler leaves threads alone. In other words, threads stay on whichever processor they start on to maintain what is called processor affinity.

A LITTLE CACHE GOES A LONG WAY

Processor affinity is an efficient scheduling practice for a number of reasons. (Affinity, in this context, refers to the marriage, so to speak, between threads and processors.) For example, affinity scheduling minimizes cache misses.

Among the more common tricks to reduce load on the processor where the thread is currently running to flush its cache. This cache flush ensures that the processor that assumes this thread's execution is able to access from RAM the most recent version of the data being processed. As you can imagine, a cache flush takes a toll on performance.

Fortunately, as mentioned earlier, NetWare 6 moves threads only when absolutely necessary. As a result, NetWare 6 minimizes the number of cache flushes. Of course, periodic cache flushes are unavoidable. After all, data in the processor cache must be copied back to normal RAM to ensure that the shared copy of the data is correct and current.

NetWare 6 uses what is called a lazy-write algorithm to copy data from processors' caches to RAM. This algorithm writes data back to RAM only when a processor's cache management circuitry recognizes that its cache memory is full and realizes that it needs to make room for new data. In this case, the circuitry flushes modified cache data back to RAM.

LOCK, BLOCK, AND BARRIER

As you can imagine, when threads from multiple programs are executing simultaneously on multiple processors, you have the recipe for an environment potentially laden with problems. For example, what happens when two threads attempt to write to the same memory space simultaneously? What can happen, without proper intervention, is that data potentially arise when threads compete for other server resources.

"In practice," says Jerome, a manager in Novell Developer Support, "multiprocessing is extremely difficult to do." Jerome is referring specifically to the tricks of the MP trade that developers of MP-enabled operating systems and programs must learn and incorporate into their programs.

A mong the more common tricks to the MP trade are synchronization primitives, which are rules for avoiding the problems inherent to multiprocessing. Developers can use these rules when writing MP-enabled programs to ensure their programs safely share resources with other MP-enabled programs.

NetWare 6 supports all of the common synchronization primitives, including, for example, mutual exclusion locks (mutexes) and semaphores. Mutexes are objects...
that ensure that only a single thread has access to a protected resource or code at any one time. Semaphores are similar to mutexes except that semaphores include counters to allow a specific number of threads access to a protected resource or code at any one time. Programs can use mutexes and semaphores to avoid data contamination, among other problems.

**All Threads Are Equal (More or Less)**

With all of these threads running on all of these processors, how does a processor know which thread to process next? To coordinate the order of thread processing, each processor calls the Scheduler’s scheduling procedures. These scheduling procedures are MP-enabled and therefore available on each processor. Hence, each processor maintains its own thread queue and manages scheduling for itself.

In fact, each processor actually maintains three queues:

- Fast work-to-do
- Normal work-to-do
- Generic thread

Of course, these aren’t official names, but they work for this discussion. The fast work-to-do and normal work-to-do queues don’t process threads, per se, but instead process what amounts to tasks.

Tasks in the fast work-to-do queue have priority over tasks in the normal work-to-do and generic thread queues. When one task is completed or relinquishes its execution, the processor first runs all tasks in the fast work-to-do queue. Fast work-to-do tasks are nonblocking, which means they never relinquish the processor but always run to completion.

Only tasks that are critical to the overall performance of the system are included in the fast work-to-do queue. For example, Hundley explains, the TCP/IP stack uses the fast work-to-do queue “to give packet processing higher priority than other activities on the server.” Giving packet processing higher priority makes sense, Hundley adds, “because processing packets is a key function of an operating system and performance is critical.”

If the Scheduler finds no tasks in the fast work-to-do queue (or if the processor has run all of the tasks in this queue), the Scheduler runs the next task in the normal work-to-do queue. Normal work-to-do tasks, unlike fast work-to-do tasks, can—and frequently do—relinquish the processor.

When tasks from the normal work-to-do queue are completed or have temporarily relinquished the processor, the processor runs the next thread in the generic thread queue. The processor processes threads in the generic thread queue in the order in which they appear (first in, first out). Most program threads line up in this generic thread queue.

**SAME MPK, MORE MP-ENABLED COMPONENTS**

If you are quite familiar with NetWare 5, you may recognize that much of what has been said about NetWare 6 is also true. In particular, NetWare 6, like NetWare 5, is a multithreaded, MP-enabled operating system. The NetWare 5 Scheduler, like the NetWare 6 Scheduler, practices affinity scheduling and moves threads from one processor to another only when necessary. In addition, NetWare 5, like NetWare 6, supports the same set of synchronization primitives.

In fact, the NetWare 6 MPK is essentially the same as the NetWare 5 MPK although, as you may expect, Novell has made a few improvements. The point is...
More Cash for Extra Cache—Is It Worth It?

Cache memory is much more expensive to produce than RAM, explains Novell engineer Greg Hundley. Consequently, each processor has only a limited amount of one to three different levels of cache memory:

- A Level 1 (L1) cache, which is typically internal to the processor chip and is "every bit as fast as the processor needs," according to Hundley
- A Level 2 (L2) cache, which is typically external to the processor chip and is "nearly as fast as the processor needs," says Hundley
- A Level 3 (L3) cache, which is also typically external to the processor chip

However useful, these comments on speed and location merely serve to point out typical cache arrangements. Speed and location don't define whether a cache is an L1, L2, or L3 cache, Hundley points out. Instead, the architectural hierarchy of a cache determines its level. That is, a cache that runs at full speed and is internal to the chip might nevertheless be an L2 cache; it depends on its relation to the other caches.

The amount of cache memory a processor has impacts its cost. For example, consider Intel's Celeron, Pentium, and Xeon processors, which are basically the same processors with less or more cache.

<table>
<thead>
<tr>
<th>Processor</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 733 MHz Celeron</td>
<td>0 KB</td>
<td>256 KB</td>
<td></td>
</tr>
<tr>
<td>Intel 733 MHz Pentium III</td>
<td>0 KB</td>
<td>256 KB</td>
<td></td>
</tr>
<tr>
<td>Intel 770 MHz Xeon</td>
<td>0 KB</td>
<td>256 KB</td>
<td></td>
</tr>
</tbody>
</table>

WHEN TCP/IP MEETS MP

The MP-enabled TCP/IP stack processes all of the packets associated with a single connection on a single processor.

How It Works

To distribute TCP/IP threads, the TCP/IP stack uses a hash of packets' source and destination IP addresses and port numbers. By using this information, the stack can ensure that the same processor handles all of the packets associated with any one TCP/IP connection.

The TCP/IP stack passes packets it receives to the application to which the packets are addressed. When the application is ready to send response data, it first checks the TCP/IP stack, informing it of the amount of data the application has to send. The TCP/IP stack, in turn, checks for a send window and, depending on the amount of send data, prepares either to send or buffer the data.

The TCP/IP stack also determines whether or not the application called TCP/IP on the same processor to which it mapped the original connection. A summary of the call in on the processor already assigned to this connection, the TCP/IP stack continues executing the application thread. If the call did not come on the processor assigned to this connection, the TCP/IP stack reassigns the request to the appropriate processor (again using a hash of the source and destination IP addresses and port numbers).

When the TCP/IP stack is prepared to send or buffer the application's response data, it checks the application's callbacks. Callbacks are sequences of instructions (called subroutines) that indicate how a program (in this case, the TCP/IP stack) should handle certain events (for example, the completion of an I/O operation). The TCP/IP stack uses callbacks to determine how to handle packets it receives and sends and is thus said to be callback driven.

In other words, services and applications that use the TCP/IP stack—including Winsock, NCPIP, Excelerator (formerly called Novell Internet Caching System [ICS]), and GroupWise—register callbacks with the TCP/IP stack. These callbacks tell the stack how to handle packets destined to or coming from these services and applications.

For example, in this case, the TCP/IP stack calls the application's callbacks to tell the application that it has prepared a send window. This call alerts the application that it can now prepare a Send
Event Control Block (ECB) to actually send the data. (ECBs are structures that control events related to the transmission and reception of TCP/IP and IPX/SPX packets in NetWare environments.)

When creating and sending the ECB, the application uses a Send Done callback. After receiving acknowledgement of the sent data, the TCP/IP stack uses a Send Acked callback to notify the sending application.

**Why You Should Care**

If you get nothing else from the previous explanation, you should understand this much: First, because the TCP/IP stack is now MP-enabled, it is no longer a bottleneck. Second, the MP-enabled TCP/IP stack can process multiple TCP/IP connections simultaneously.

Third, the MP-enabled TCP/IP stack processes all packets and callbacks associated with a given connection on the same processor—and this is good news for several reasons.

For one thing, and as you can probably guess, by processing connections on the same processor, the TCP/IP stack minimizes the number of cache misses. The MP-enabled TCP/IP stack also eliminates the possibility of out-of-order processing (which is possible in any multiprocessing environment). If different processors handle packets and callbacks from the same TCP/IP connection, the stack may end up processing packets and callbacks out of order. Fortunately, the TCP/IP stack’s approach to processing ensures this problem does not occur.

By assigning one connection per processor, the MP-enabled TCP/IP stack also avoids race conditions. Race conditions are situations where the order in which tasks are processed changes the results. (For more information, see “On Your Mark, Get Set, No! Avoiding Race Conditions” on p. 12.)

Finally, because all operations related to a single TCP/IP connection remain on a single processor, the NetWare 6 TCP/IP stack creates what feels like a single-processor environment for that connection—and the NetWare 6 TCP/IP stack does so without the use of other synchronization techniques.

**MAKING THE MOST OF MP**

Running NetWare 6—with all of these newly MP-enabled components—

---

**CLIB Versus NKS/LibC**

Applications written to the new Novell Kernel Services (NKS) Application Programming Interface (API) set make better use of the available processors in a multiprocessing environment than applications written to the CLIB API set. Why? The simple answer is that CLIB funnels to Processor 0, whereas NKS/LibC does not.

Since you probably want more information about the differences between CLIB and NKS/LibC, a more detailed answer follows. (This discussion is based on information provided by Novell engineer Tom Buckley)

**CLIB**

Suppose an application written to CLIB wants to communicate with a remote server. To do this, the application first uses the CLIB Requester to make a call to the Directory Services APIs (DSAPIs). To open a connection to the remote server, the DSAPIs send the server a NetWare Core Protocol (NCP) packet.

The execution of this request eventually trickles down to the CLIB Requester’s packet-sending function, which packages the request into a Send Event Control Block (ECB). (ECBs are structures that control events related to the transmission and reception of TCP/IP and IPX/SPX packets in NetWare environments.)

The calling thread (that is, the thread that called the DSAPIs in the first place) then funnels to Processor 0, sends the packet, and “goes to sleep” (in other words, suspends itself) while waiting for a reply.

When the reply from the remote server arrives, the server thread processes the reply hands the ECB to the TCP/IP stack, which in turn forwards to the server a NetWare Core Protocol (NCP) packet. Instead, the calling thread sends the packet and suspends itself, all the while remaining on the same processor from which the calling thread placed the call.

**NKS/LIBC**

In comparison, suppose an application written to the NKS APIs wants to communicate with a remote server. To do this, the application makes a call to NKS/LibC to initiate an NCP packet send. The execution of this call trickles down to NKS/LibC’s packet-sending function. The packet-sending function packages the request into a Send ECB and sends the packet. The calling thread then goes to sleep while it awaits a reply.

When the reply from the remote server arrives, the server thread that processes the reply hands the ECB to TCP/IP, which in this case forwards the ECB to Winsock. (Winsock is a specification that basically provides a programmer-friendly interface to TCP/IP.)

WinSock then calls a subroutine registered to Winsock by the new NKS NCP Client (which runs on the server). All applications that use Winsock, including the NKS NCP Client, register subroutines. These subroutines (also called callbacks) are sequences of instructions that tell Winsock how to handle certain events, such as the arrival of new data.

Hence, in this case, Winsock calls an NKS NCP Client subroutine to inform the client that the socket has new data. The NKS NCP Client then validates the reply packet and awakens the original calling thread. This thread continues to check the packet for accuracy and then deactivates, returning to the application that originally called the NKS/LibC.

**WHAT’S THE DIFFERENCE?**

Did you catch the big difference between applications written to CLIB and those written to NKS/LibC? Unlike threads that call the CLIB Requester to initiate an NCP packet send, threads that call the NKS/LibC do not have to funnel to Processor 0 to send the NCP packet. Instead, the calling thread sends the packet and suspends itself, all the while remaining on the same processor from which the calling thread placed the call.

When this calling thread is later awakened, it does not need to migrate off of Processor 0 because it was never on Processor 0. Instead, the calling thread remains on the same processor while returning to the application from whence the calling thread came.
**Coming Soon**

The shipping version of GroupWise 6 is already MP-enabled to the degree that it can be—but will soon make better use of available processors than it does now. The Post Office Agent (POA) and Message Transfer Agent (MTA) in the shipping version of GroupWise 6 are MP-enabled. However, they are dependent on CLib and, consequently, funnel to Processor 0 for I/O functions. When you use GroupWise 6, you will experience a performance gain of about 15 percent to 20 percent with one additional processor, says product manager Howard Tayler.

A team of Novell engineers is currently rewriting GroupWise 6 to the Novell Kernel Services (NKS) Application Programming Interface (API) set, which is already MP-enabled to the degree that it can be. However, they are dependent on CLib and, consequently, funnel to Processor 0 for I/O functions. When you use GroupWise 6, you will experience a performance gain of about 15 percent to 20 percent with one additional processor, says product manager Howard Tayler.

Henriksen points out that running NetWare 6 without also running MP-enabled applications is beneficial, but admittedly minimal. "When portions of the operating system—such as LAN and Disk—are moved from Processor 0 to other processors, this frees up cycles for use by non-MP-enabled applications that run on Processor 0."

Hence, theoretically, you should still experience a performance increase. Nevertheless, Henriksen and other Novell engineers agree, by running NetWare 6 without also running MP-enabled applications, you benefit "less than you would if you ran M P-enabled applications."

A mong other applications, GroupWise 6 runs on NetWare 6. The current version of GroupWise 6 makes some use of the NetWare 6 multiprocessing environment. Shortly after the upcoming release of NetWare 6, Novell will provide an update for GroupWise 6. This update will enable GroupWise 6 to make full use of the NetWare 6 multiprocessing environment. (For more information, see "Coming Soon.")

Novell now recommends for developers who want to correctly MP-enable their applications. Novell engineer Jay Parker says that while the work involved is not technically difficult, the team still has a lot of work to do. Nevertheless, the team hopes to have the NKS update ready for the GroupWise 6 Service Pack, which should be available within two months of NetWare 6.

The end result of the team’s effort will not be trivial: Because the newly MP-enabled version of GroupWise 6 will be written to the NKS API set, the POA and MTA will no longer have to funnel to Processor 0. Consequently, when you run the upcoming Service Pack release of GroupWise 6 on a NetWare 6 server, you can expect to see a performance gain of a full 95 percent, says Tayler.

The similarities end there, however. The NKS APIs include many significant new interfaces that replace CLib’s entire threading model. A s Novell engineer Russell Bateman points out, "NKS promotes a set of threading interfaces that are more sophisticated and correct than those in CLib." (See "The Future of Application Development on NetWare with NLMs," AppNotes, Sep. 1999. You can download this article from http://developer.novell.com/research/devnotes/1999/september/03.)

As a result, an application written to NKS APIs will make better and more complete use of available processors than applications written to CLib. Why? The simple answer is because threads that call the CLib I/O routines funnel to Processor 0. In contrast, threads that call the NKS I/O routines— or any other routines—remain on whatever processor they start on. (For a more complete explanation, see "CLib Versus NKS/LibC" on p. 16.)

Writing applications to NetWare are using the NKS APIs, is "nothing less than an exercise in writing correctly multithreaded code—just as one would to any other platform," says Bateman. (Bateman also maintains that MP-enabled programs are simply correctly written multithreaded programs.) Given this statement, you should not be surprised to learn that porting applications written to the NKS API set from one platform to another is relatively simple.

For example, developers who write to the NKS APIs will have considerably less work to do than developers who write to the older CLib APIs when it comes time to port their applications from NetWare 6 to NetWare 5 to NetWare’s code-named Modesto. (Modesto is Novell’s next-generation operating system platform. In fact, the NKS API is the actual interface to the Modesto kernel.)

**TOOLS FOR WRITING MP-ENABLED APPLICATIONS**

Why does the current version of GroupWise 6 make only limited use of the NetWare 6 MP environment? And how will the updated version make full use of this environment? The answers to both questions have to do with the Application Programming Interface (API) set to which the application has been, and will be, written.

The existing version of GroupWise 6 was written to the CLib API set. The update, in contrast, will be written to the Novell Kernel Services (NKS) API set. Trial versions of the NKS API set have been available as part of the Novell Developer Kit (NDK) for quite some time. (These trial versions are located in theNKS APIs, is "nothing less than an exercise in writing correctly multithreaded code—just as one would to any other platform," says Bateman. (Bateman also maintains that MP-enabled programs are simply correctly written multithreaded programs.) Given this statement, you should not be surprised to learn that porting applications written to the NKS API set from one platform to another is relatively simple.

For example, developers who write to the NKS APIs will have considerably less work to do than developers who write to the older CLib APIs when it comes time to port their applications from NetWare 6 to NetWare 5 to NetWare’s code-named Modesto. (Modesto is Novell’s next-generation operating system platform. In fact, the NKS API is the actual interface to the Modesto kernel.)

**CONCLUSION**

NetWare 6 will be available in public beta by May and on the shelves during the third quarter of this year. You realize what this means, don’t you? You’ve just experienced a first: For what we’re willing to bet is the first time in your life, you have taken something apart to learn how it works—before ever even using it.

Linda Kennard works for Niche Associates, a technical writing and editing firm located in Sandy, Utah.

**FEATURE NetWare 6 and MP**