

Meeting the bandwidth challenge: Building Scalable Networking Equipment Using SMP

In their efforts to build a comprehensive range of networking products, many equipment manufacturers have invested in an equally wide range of operating systems (OSs). The results are predictable: code can't be reused across products, engineers can't move quickly from one project to another, and the networking products themselves can't offer end-to-end consistency of software services and management tools - much to the customer's inconvenience. In this paper, we look at how a microkernel OS based on network-transparent IPC can address these issues by allowing applications to be coded once, then deployed across entire product lines. With this OS architecture, the same application can run on a single-processor device, be partitioned across networked processors, or run on an SMP system, all without recoding or relinking. The net effect: less development effort, reduced testing, greater product consistency, and higher return on investment.

INTRODUCTION

The bandwidth "explosion" is putting extreme demands on the control-plane processor of today's core routers and switches, to the point where even the fastest CPU can't keep pace. In a high-end router, for instance, the control-plane CPU must now maintain a routing table of 500,000 or more entries, using compute-intensive protocols such as OSPF. At the same time, the CPU must also process SNMP packets, perform OA&M functions, support operator consoles, download a subset of the routing table to each line card - the list goes on.

With network bandwidth doubling about twice as fast as CPU performance, the problem shows no sign of letting up. Consequently, more and more systems designers are choosing to distribute the workload across multiple CPUs, using symmetric multi-processing (SMP).

SMP is often called the "shared everything" approach to multi-processing, since the multiple CPUs share equal access to the same memory array, the same operating system, and the same I/O (see Figure 1). Nonetheless, each CPU in an SMP system can act independently. For instance, if the routing database is multithreaded, one CPU could update the database while a second CPU handles a database lookup. And while those database threads are executing, other CPUs could be handling exception packets, system administration tasks, and so forth.

As a means of boosting performance, SMP holds several attractions, including relatively low cost. For instance, to scale from one CPU to two CPUs, the incremental cost - in terms of dollars, board space, and power consumption - is only that of the extra CPU. There's no need to pay for extra bridge chips or support chips, as you would if you distributed the workload across a networked cluster of loosely coupled CPUs. What's more, you can add the extra CPU without tak-

ing up an additional slot in the chassis, thus providing headroom for future growth.

That said, SMP isn't a silver bullet. For instance, the law of diminishing returns comes into play as you add more CPUs, since they must all contend for the same memory subsystem. Hence it's critical that the operating system (OS) used to implement SMP doesn't add any unnecessary overhead on top of these natural barriers. That's a problem, since SMP is commonly associated with large, monolithic OSs used in enterprise server roles. Because the kernels in these OSs contain the bulk of OS services, adding support for SMP typically requires large numbers of performance-robbing modifications and spinlocks throughout the kernel code. And, since all device drivers run in the kernel space, adding SMP support means modifying each driver as well.

In fact, one reason SMP isn't used more frequently is the difficulty of implementing it in software. As a result, systems designers must often deploy limited implementations, where the OS and drivers run on one CPU,

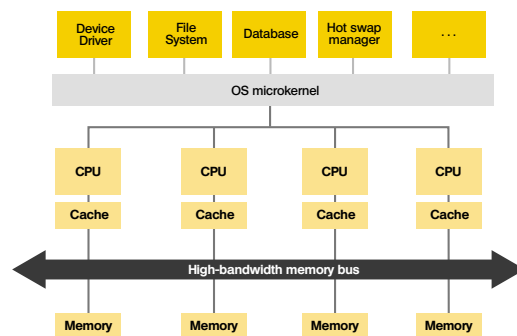


Figure 1. In a true SMP implementation, any kernel thread, any process (e.g. database, file system), and any thread within a process can be scheduled for execution on any CPU.

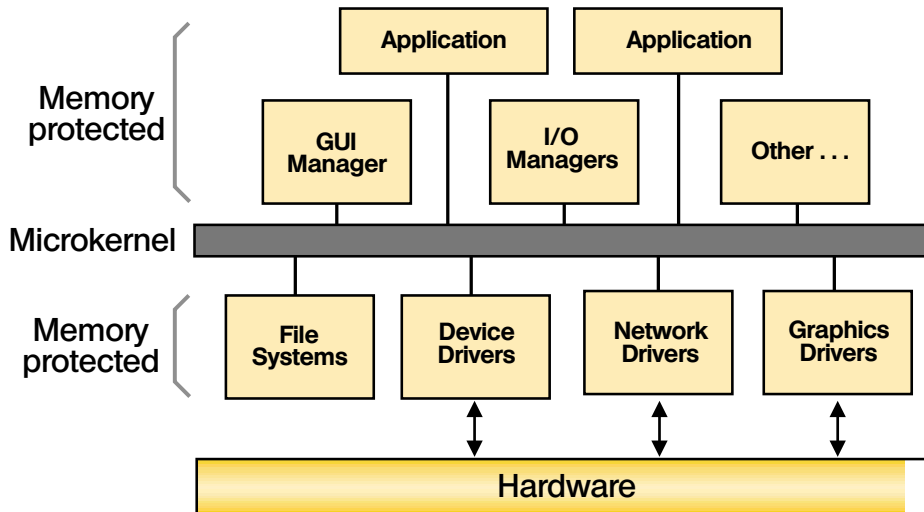


Figure 2. In a microkernel OS, supporting SMP requires only a few additional kilobytes of kernel code, since only core OS services (e.g. thread scheduling, interrupt handling) reside in the kernel. All other multithreaded services can take full advantage of SMP without code changes.

while only specific, hand-tuned applications run on the remaining CPUs. With this two-tiered approach, the SMP-enabled applications often don't have full access to OS services. Meanwhile, all non-SMP-enabled services must compete for a single CPU.

NO RECODING REQUIRED

An OS with a microkernel architecture, such as the QNX RTOS, helps designers avoid the above problems. Compared to monolithic OS kernels, the QNX microkernel is extremely small, since most OS-level services (file systems, drivers, protocol stacks, and so on) exist as user programs that run outside the kernel space (see Figure 2). Consequently, the kernel modifications required for SMP are equally small: just a few additional kilobytes of code. In fact, only the kernel has to be modified. All other multithreaded processes - file systems, drivers, applications - can gain the performance advantages of SMP without the need for code changes.

Since so little code has to be added to the kernel, this approach to implementing SMP incurs negligible overhead. And compared to monolithic OS models, it's inherently more reliable, since there's simply less to go wrong. Just as important, this approach allows customer-written applications and drivers - not just modules supplied by the OS vendor - to move unmodified between single-processor and SMP systems.

For instance, a router manufacturer can develop an application or driver once, then deploy the same binary across a product line that includes both single-processor and SMP systems. Alternately, the manufacturer could ship an SMP-ready device with one CPU installed, and the end-user could then simply plug in more CPUs as performance demands increase; no need to change software. This "scalability on demand" provides a simple, and cost-effective, migration strategy for any user whose networking requirements are growing rapidly.

GETTING IN SYNC

By allowing CPUs to act asynchronously and independently, SMP offers the systems designer a lot of flexibility. But this flexibility can come at a price. Without careful use of synchronization primitives, processes running on different CPUs can end up in deadlocks (where one or both processes wait for a resource that the other has locked) or in race conditions (where multiple processes can access the same resource at the same time, with unpredictable results). These problems can occur on a single-processor device, but are much more likely to show up in an SMP system.

Again, a microkernel OS can help address these problems. For instance, most processes and threads in a microkernel OS communicate through message passing, which not only lets processes exchange data, but also provides a built-in mechanism to keep processes synchronized. For instance, say you have two processes: A and B. If A sends a message to B, A will automatically stop running, or become blocked, until it has received a reply from B. This ensures that the processing performed by B for A is complete before A can resume executing (see Figure 3).

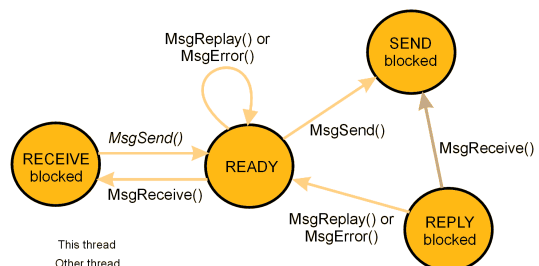


Figure 3. Message passing simplifies the synchronization of threads in an SMP system, since the act of sending a message automatically causes the sending thread to be blocked and the receiving thread to be scheduled for execution.

The software developer doesn't have to write any special code to implement this built-in synchronization, which in many instances can render hand-coded synchronization services - and the complexity of debugging them - unnecessary. In fact, message passing itself requires no special programming in a microkernel OS like QNX, but is implemented using standard POSIX calls. The actual job of passing messages between processes is handled transparently by the underlying C library.

Unfortunately, some believe that message passing comes at a price, in terms of increased overhead. But, in fact, message-delivery performance can approach the memory bandwidth of the underlying hardware, if implemented correctly. The QNX RTOS, for example, employs various techniques - such as multipart messages and the copying of message data directly from thread to thread - to ensure that performance remains on a par with conventional IPC methods. In addition, the OS will always try to schedule sender and receiver threads on the same CPU to ensure that their data stays in the processor's high-speed local cache.

HARD OR SOFT AFFINITY?

Which brings us to our next point. To achieve optimal SMP performance, some OSs support processor affinity: the OS scheduler will always try to dispatch each thread to the CPU where the thread last ran. This way, the CPU can often fetch the thread's instructions directly from cache, rather than having to reload the instructions from the system's main memory. Without processor affinity, threads can randomly drift from CPU to CPU, overwriting each other's cached instructions and forcing the cache to be continuously reloaded.

Nonetheless, in some cases, such as a busy router with many active threads, a "soft" form of processor affinity can't always ensure that a thread will be rescheduled on the same CPU. Hence the QNX RTOS also provide "hard" processor affinity, which lets you lock a thread to one or more specific processors. To do this in QNX, you use an affinity mask, which is simply a bitmap: each bit position indicates a processor on which a thread is allowed to run (see Figure 4).

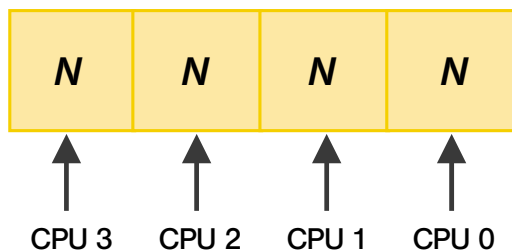


Figure 4. Affinity mask for a quad SMP system.

For instance the bitmap (hex mask value 0x01) in Figure 5 would run a thread on CPUs 0 or 2.

Likewise a bitmap of 0100 (hex mask 0x04) would run the thread on CPU 2; a bitmap of 0111 (hex mask 0x07) would run the thread on CPUs 0, 1, or 2; and so on.

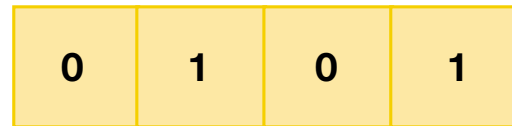


Figure 5. Run the thread on CPUs 0 and 2.

THE RELEVANCE OF REAL TIME

By careful use of the affinity mask, a systems designer can optimize runtime performance by, say, locking non-realtime processes to a specific CPU. The remaining CPUs would then remain free to execute time-critical processes. In general, however, this approach isn't necessary if you use a true RTOS, since the OS scheduler will always preempt a lower-priority thread immediately when a higher-priority thread becomes ready.

Thanks to these preemptive capabilities, an RTOS can help an SMP device handle increases in system load without the cost or complexity of adding more CPUs. That's because an RTOS always schedules time-critical tasks, such as routing table updates, in a predictable time frame, no matter how many other processes demand CPU time. Response times can remain constant, even as overall system load increases. Also, in some RTOSs, context-switch speeds for threads and processes are often in the submicrosecond range - orders of magnitude faster than in monolithic OSs conventionally used for SMP. As a result, the CPUs waste much less time switching from one thread to another and have more time to execute compute-intensive applications.

CLUSTERS BREAK BOTTLENECKS

As you add CPUs to an SMP system, the increased traffic on the shared-memory bus can become a bottleneck, limiting system throughput. So, rather than add more CPUs to the same SMP card, designers of very high-end devices can distribute the workload across a "loosely coupled" cluster of SMP cards, where each card has its own OS, memory array, I/O, and so on (see Figure 6). With this approach you could, for instance, implement load-balancing software that would distribute work to the card that currently has the most available compute cycles. And there's another advantage to this "shared nothing" model of multiprocessing - higher availability. If one SMP card experiences catastrophic failure, the remaining cards can take over all or some of that board's duties until it is replaced or restarted.

Nonetheless, attempting to distribute applications across these networked cards poses its own challenges. For instance, conventional OSs don't provide network-transparent interprocess communication (IPC). So, if you split up an application's components across different cards, you must also add network-specific code so those components can continue talking to each other. As an added complication, drivers and protocols in most OSs are bound to the kernel. To move any of these services from one SMP card to another, you typically have to create and test new kernel

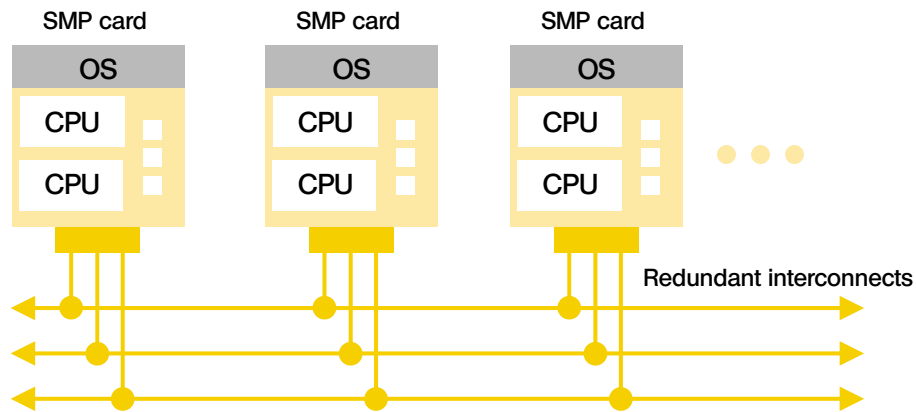


Figure 6. In a loosely coupled cluster of SMP cards, each card has its own resources (OS, memory, I/O, etc.), providing higher availability and virtually unlimited scalability.

images, one for each board.

As a message-passing microkernel RTOS, QNX side-steps these problems in two ways. First, applications, protocols, drivers, and even high-level OS services (e.g. file systems) are all decoupled from the OS kernel. As a result, each of these modules is an independent, MMU-protected process whose binary can be moved easily from one CPU card to another. No kernel re-configuration required. Second, the message passing that processes use to communicate with each other also shields them from networking issues. For instance, when process A sends a message to process B, it doesn't have to know which node (i.e. SMP card) that B resides on - if B is local, the microkernel will route the message directly; if B is on another node, then a separate network manager can forward the message to that node. With this approach, neither A nor B has to invoke any special code to communicate across networked SMP cards. Consequently, programmers can design an application just once - there's no need to recode and retest the application if it's subsequently moved or partitioned across multiple boards or CPUs (see Figure 7).

FREEDOM TO CHOOSE

Share everything or share nothing? When it comes to multiprocessing, the SMP and clustering models both have their advantages. For the systems designer, the real issue is being free to choose the best model for the job at hand, and to combine the models easily when both are needed. This ease of scalability isn't merely desirable. It is, in fact, an immense commercial advantage - whether you consider development costs, time-to-market, or, ultimately, the satisfaction of customers who demand network solutions that can handle more services on demand and at the lowest possible cost ■

Paul Leroux is a technology analyst with QNX Software Systems, where he has served in various roles since 1989. Paul performs extensive research on OS architecture issues, with an emphasis on applications for information appliances and networking equipment.

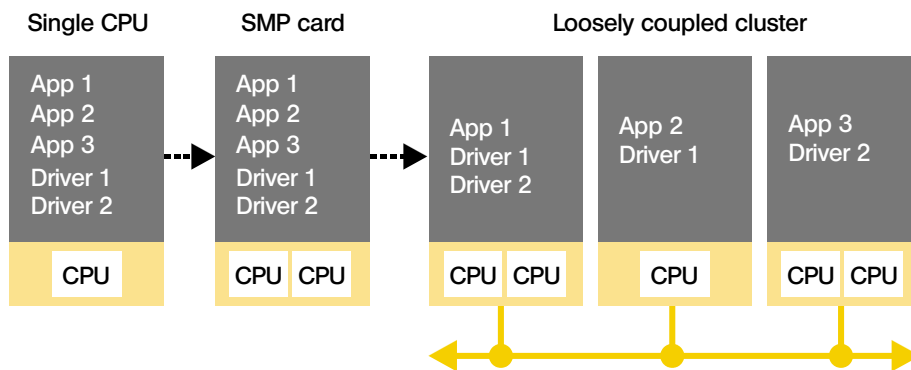


Figure 7. With a microkernel OS architecture, drivers, protocol stacks, and OS modules can migrate from a single-CPU device to an SMP system - or be distributed across a network of loosely coupled devices - without recoding.