MICHAEL W. LUCAS

FreeBSD 5 SMPng

THE NETWORK STACK

Michael W. Lucas is a network consultant and author of *Absolute BSD, Absolute OpenBSD, Cisco Routers for the Desperate*, and the forthcoming *PGP & GPG*. He has been logging onto UNIX-like systems for twenty years and finds lesser operating systems actively uncomfortable.

mwlucas@blackhelicopters.org

The author wishes to gratefully acknowledge Robert Watson's strong contribution to this article.

FREEBSD IS ONE OF THE GRANDPARENTS of open source operating systems, and FreeBSD version 4 is considered the gold standard of high performance by its user community. In this article we'll discuss the improvements in FreeBSD 5, using the network stack as an example of the particularly heinous problems faced when enhancing multiprocessor operating systems.

FreeBSD 5 had many disruptive new features (such as the GEOM disk layer and ACPI) but also had extremely high ambitions for its new SMP implementation. This SMP infrastructure was necessary for the future growth of FreeBSD, but required massive rewrites in many parts of the system. The new multi-CPU project, dubbed "SMPng," steered FreeBSD in a direction that promised incredible performance enhancements—at the price of a lot of work.

To understand why this was considered worthwhile, we need to consider some basics of multiprocessor computing. What we usually think of as "multiprocessing" is actually "symmetric multiprocessing," or SMP. In SMP you use multiple general purpose processors, all running the same OS. They share memory, I/O, PCI busses, and so on, but each CPU has a private register context, CPU cache, APIC timer, and so on. This is certainly not the only approach: all modern video cards have a Graphics Processing Unit (GPU), which could be considered a special purpose asymmetric multiprocessor. Managing multiple identical general purpose CPUs has become dramatically more important with the advent of multi-core CPUs.

A multiprocessor-capable OS is one that operates correctly on multiprocessor systems. Most operating systems are designed to give the user access to those extra CPUs simply as "more computing power." While many people have implemented alternatives to this, the general idea that more CPU means more horsepower is still what most of us believe.

Additionally, adding processors can't be allowed to change the look-and-feel of our operating system. Managing these processors becomes much simpler if you abandon the current process model, standard APIs, and so on. Many of these APIs and services were designed for systems with only a single CPU and assumed that the hardware had only one processor executing one task at a time. As with so many other hardware evolutions, it would be easier to knock down the house of UNIX and build a strip mall. Instead, the FreeBSD Project has had contractors climbing over the old house to bring it up to today's building code—and gain some extras while we're at it.

Obviously, the goal of adding processors is improving performance. The problem is that "performance" is a very vague term: it depends on the work you're doing, and trade-offs happen everywhere. There's a 100-mpg carburetor that works wonderfully, if you don't mind doing 0 to 60 in about an hour.

The best way to handle performance tuning in SMP is to measure how your system performs under the workload you're interested in, and continue measuring as you add additional CPUs. The SMP implementation needs to not slow down the application in itself, and then it needs to provide features to make it possible to accelerate the application. In an ideal world, your application performance would increase linearly with additional processors—an eight-CPU system would perform eight times as well as a one-CPU system. This simply isn't realistic. The OS and application will be slowed by having to share resources such as memory and bus access, and keeping track of where everything is becomes increasingly complex as the number of CPUs increases. Consistent measuring and benchmarking are vital when embarking on an SMP implementation.

Implementing SMP

Implementing SMP is simple. First, make it run. Then make it run fast. Everything else is just petty details.

The obvious place to begin is in the kernel. Nothing happens until your kernel notices the additional processors. Your OS must be able to power up the processors, send them instructions, and attend to everything that makes it possible to use the hardware.

Then your applications must be able to use the additional CPU. You might find that your important application can only run on one processor at a time, rendering that additional processor almost useless. You can have an application monopolize one CPU while the other CPU handles all the other petty details of keeping the system up, but this is less than ideal.

Your OS libraries and utilities play a vital part in this. Perhaps the most common way to make an application capable of using parallelism is by using threads. Your OS must include a multiprocessor-capable and well-optimized threads library. It can do its thread support in either the kernel or userland. Some applications use more brute-force methods of handling parallelism—the popular Apache daemon forks copies of itself, and those children can automatically run on separate processors.

Once your application can use additional processors, start measuring performance. By observing the system as it handles your work you can identify and address the bottlenecks in your real-world load. Once you fix those bottlenecks, benchmark again to find the new bottleneck. Shuffle your trade-offs until you reach an acceptable average for all of your workloads.

Traditional OS kernels expect that there is only one CPU, and so when the kernel returns to a task, all of the appropriate resources should be right where they were left for that task. When the machine has multiple processors, however, it's entirely possible for multiple kernel threads to access the same data structures simultaneously. Those structures can become corrupted. This makes the kernel angry, and it will take out its feelings on you one way or another.

You must implement a method of maintaining internal consistency and data synchronization, and provide higher level primitives to the rest of the system. Some applications require that certain actions be handled as a whole (known as "atomic operations"). Others simply insist that certain things are done before other things. Your synchronization model must take all of this into account, without breaking the API so badly that you scare off your users. Your choice of locking model affects your system's performance and complexity, and so is very important. For example, let's contrast the locking model used with FreeBSD 3.x/4.x to that used in 5.x and later.

The Big Giant Lock (BGL) model used in FreeBSD 3.x/4.x is the most straightforward way to implement SMP. The kernel is only allowed to execute on one CPU at a time. If a process running on the other CPU needs to access the kernel, it is held off (spinlocks) until the kernel is released by the other process.

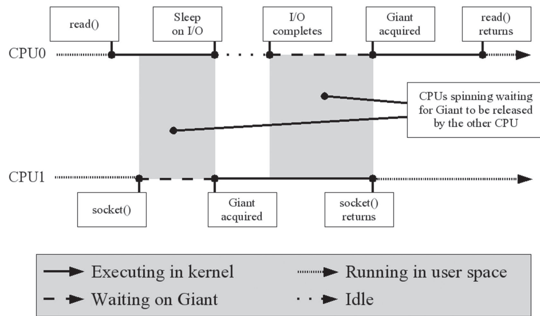
Contention occurs in the BGL model when tasks on multiple CPUs compete to enter the kernel. Think about how many types of workload access the kernel: user threads that do system calls, interrupts or timer driver activity, reading or writing to disk or networks, IPC, scheduler and context switches, and just about everything else. On a two-CPU system this isn't too bad—at least you're doing better than you would with one CPU. It's horrible on a four-CPU system, and unthinkable on an eight-way or bigger.

The locks are obviously necessary for synchronization, but the costs are high. Overall, a dual-CPU system is an improvement over a single processor.

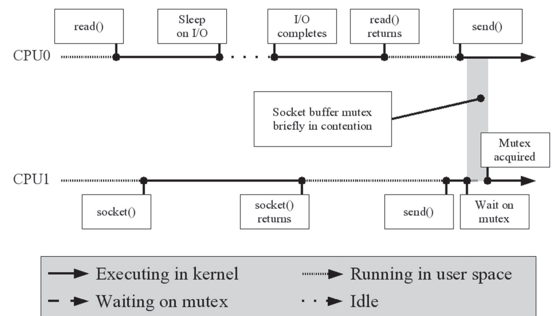
Fine-Grained Locking

FreeBSD 4.x's Big Giant Lock was the main performance bottleneck, and just had to go. That's where fine-grained locking came in. Fine-grained locking is simply smaller kernel locks that contend less. For example, a process that has entered the kernel to write to a file shouldn't block another process from entering the kernel to transmit a packet. The FreeBSD developers implemented this iteratively. First they locked the scheduler and close dependencies such as memo-

Context Switching in a
Giant-Locked Kernel



Context Switching in a
Finely Locked Kernel



ry allocation and timer events. High-level subsystems followed, such as a generic network stack lock or a file-system lock. They then proceeded to data-based locking. Once they hit this point, it was a simple matter of watching to identify the new bottlenecks and lock them more finely.

Goals along the path include adopting a more threaded architecture and implementing threads where the kernel can work in parallel. Interrupts in particular were permitted to execute as threads. FreeBSD also had to introduce a whole range of synchronization primitives such as mutexes, sx locks, rw locks, and semaphores. Low-level primitives are mapped into higher level programming services. Atomic operations and IPIs are at the bottom, which are used to build mutexes, semaphores, signals, and locks, which, in turn, are assembled into lockless queues and other structures at the very top.

As this was a gradual migration rather than an all-at-once conversion, subsystems that were not yet properly locked during the conversion were allowed to "seize" the Giant Lock. A device driver that had not yet been converted was allowed to scream "OK, everybody out of the kernel, I must process an interrupt!" This was called "holding" the Giant Lock, and it reduced performance to the 3.x/4.x level.

One by one, each system was finely locked, the BGL slid off, and newly exposed problems resolved. This produced a very different contention pattern.

This was complicated, of course, by the fact that FreeBSD is a project in use by millions of people around the world. People even consider selected points along the development version stable enough for production use. If the FreeBSD team could have simply declared, "The development branch of FreeBSD will be utterly unusable for six months," fine-grained locking could have been accomplished more quickly. They would have also alienated many users and commercial sponsors. While commercial companies can get away with this, a project like FreeBSD simply can't. The team did the equivalent of changing a car's oil while said vehicle was barreling down the freeway at 80 miles an hour.

Today, FreeBSD 5.x has fine-grained locking in most major subsystems, except for VFS. The network stack as a whole runs without Giant, although a few network protocols still require it. Some high-end network drivers execute without seizing Giant. FreeBSD 6.0 (which should be out by the time this article reaches print) is almost completely Giant-free. VFS itself runs without Giant, although some file systems do not. (Those of you running high-performance databases on a FAT file store, with a server using those dollar-a-dozen Ethernet cards, might not be pleased with its performance.) A few straggling device drivers require the BGL, but those are slated for conversion or execution before FreeBSD 7.0 is released (probably in 2007). The network stack also runs without the BGL. As locking the network stack was one of the more interesting parts of implementing fine-grained locking, let's take a closer look at it.

Locking the Network

The FreeBSD network stack includes components such as the mbuf memory allocator, network device drivers and interface abstractions, a protocol-independent routing and event model, sockets and socket buffers, and a slew of link-layer protocols and network-layer protocols such as IPv4, IPv6, IPSec, IPX, EtherTalk, ATM, and the popular Netgraph extension framework. Excluding distributed file systems and device drivers, that's about 400,000 lines of code. To complicate things further, FreeBSD's TCP/IP stack has been considered one of the best performers for many years. It's important not to squander that reputation!

Locking the network stack has very real problems. Overhead is vital: a small per-packet cost becomes very large when aggregated over millions of packets per second. TCP is very sensitive to misordering, and interprets reordered packets as requiring fast retransmit. Much like our 100-mpg carburetor, different op-

timizations conflict (e.g., optimizing for latency can damage throughput).

FreeBSD uses a few general strategies for locking the network stack. Data structures are locked. Also, locks are no finer than that required by the UNIX API—e.g., parallel send and receive on the same socket is useful, but not parallel send on the same socket. References to in-flight packets are locked, not the packets themselves. Layers have their own locks, as objects at different layers have different requirements.

Locking order is vital. Seizing locks incorrectly can cause deadlocks. Driver locks are leaf locks. The network protocol drives most inter-layer activity, so protocol locks are acquired before either driver locks or socket locks. FreeBSD 5 avoids lock problems via deferred dispatch.

Transmission is generally serial, so the work is assigned to a single thread. Reception can be more parallel, so work can be split over multiple threads.

Increasing Parallelism

All of the above is just "making it run." Afterwards came time to "make it run fast." Once the network stack is freed of the Big Giant Lock, pick an interesting workload and see where contention remains. Where was CPU-intensive activity serialized in a single thread, causing unbalanced CPU usage? Identifying natural boundaries in processing, such as protocol hand-offs, layer hand-offs, etc., both restricted and inspired further optimizations. Every trade-off had to be carefully considered and then tested to confirm those ideas. Context switches and locks are expensive, so they had to be made as useful as possible.

All this had its own challenges. The FreeBSD-current mailing list (for those people using the development version) saw many reports of deadlocks, poor performance under edge situations, and any sort of weird issue imaginable. While FreeBSD's sponsors were very generous with donations of test facilities, no test can possibly compare with the absurd range of conditions found in the real world.

One not uncommon problem during development was deadlock. If threads one and two both require locks A and B, and thread one holds A while thread two holds B, the whole system grinds to a halt. This deadly embrace was avoided by a hard lock order on most mutexes and sx locks, disallowing lock cycles, and the WITNESS lock verification tool. There's also a variable, hierarchal lock order. Lock order is a property of data structures, and at any given moment the lock order is defined for that data structure. The lock order can change as the data structure changes. And a master lock serializes simultaneous access to multiple leaf locks. Ordering was vital to avoiding deadlock—but weakening ordering can improve performance in certain cases.

Awareness of locking order and violations is critical throughout this. The WITNESS run-time lock-order monitor tracks lock-order acquisitions, builds a graph reflecting the current lock order, and detects lock-order cycles. It also confirms that you're not recursively locking a non-recursive lock as well as detecting other basic problems. WITNESS uses up a lot of CPU time but is invaluable in debugging.

Every lock is another slice of overhead. FreeBSD 5.x amortizes the cost of locking by avoiding multiple lock operations when possible, and it amortizes the cost of locking over multiple packets. When possible, locks are coalesced or reduced. Combining locks across layers can avoid additional locks. If you lock finely enough you can cause "live lock," where your system is so busy locking and unlocking from interrupts that it does no actual work.

Some workloads handled parallelization better than others. Parts of the network stack, such as TCP, absolutely require serialization to avoid protocol performance problems. Any sort of naive threading violates ordering. FreeBSD uses two sorts of serialization: thread serialization and CPU serialization, which uses per-CPU data structures and pinning/critical sections.

Today's SMPng Network Stack

FreeBSD 5.x and above largely run the network stack without the Big Giant Lock, and 6.x shows substantial improvements over both 4.x and 5.x. The project has progressed from raw functionality to performance tuning. The development team is paying close attention to the performance of popular applications such as MySQL, as well as basic matters such as raw throughput.

Many workloads, such as databases and multi-threaded/multi-process TCP use, show significant improvements. The cost of locking hampers per-packet performance on very specific workloads, such as the packets per second when forwarding and bridging packets. And, compared to the gold standard of 4.x, performance on single-processor systems is sometimes suboptimal. While single-processor performance is being carefully monitored and enhanced, as dual-core systems become the standard even on workstation systems this will be less important. The FreeBSD team is actively working on performance measurement and optimization.

Juggling all these optimizations is hard; it took about five years to get past merely functional to optimal. The oil change at 80 mph is just about done, though, and it's time to floor the accelerator and see what this baby can do.