Breaking the Big Lock:
A Look at Symmetric Multiprocessing in UNIX

By Ron Overton

Many computer architectures have had versions of UNIX developed for them that implement symmetric multiprocessing, commonly denoted SMP. This paper attempts to examine the state of SMP in UNIX systems for the x86 architecture. Running UNIX on x86 architecture is a relatively new development in the UNIX world which began somewhere around the time Intel introduced the 386 in 1985. The Intel 386 was Intel's first 32 bit processor and had the capability to execute in protected mode. Also, because of the late introduction of multiple processors to the x86 architecture, SMP development for this platform is lively and being explored rigorously by commercial UNIX vendors and volunteer projects. First, it is important to take a look at the traditional UNIX model and how it relates to SMP development. Then, an in depth examination of the workings behind ongoing development in a popular volunteer project will provide a view into the present state and future of SMP development.

"In the UNIX model, the main clients for resources are processes and interrupt handlers" (Lehey, 1). Interrupt handlers always run in kernel space and usually run on behalf of the system. Processes on the other hand can run in either kernel or user space. Processes running in user space use interrupt handlers as an entry point into the kernel to run kernel code. When this event occurs the process is running in the kernel. In the traditional UNIX model, several assumptions are made about how processes and interrupt handlers get run.

The first assumption is that there is only one CPU and that it is shared. Also, interrupt handlers are always given a higher priority than processes and there is a separate priority hierarchy that allows higher priority interrupts to preempt lower priority interrupts. In addition the scheduler acts in a very predictable manner. It is run whenever a process voluntarily gives up the CPU, if the processes time slice expires, or if a process is preempted by a higher priority process or interrupt. If the process was preempted out of the CPU while running in kernel mode the scheduler does not run until it returns to

user mode and exits the kernel or the process sleeps.

All processes running in the kernel share the same kernel address space. Therefore, it is possible that if a process is preempted while running in the kernel, another process could enter the same routines in the kernel and wreak havoc on the structures that were previously being used by the preempted process. This follows from the rule that the scheduler will only run if a process in kernel space exits to user space or puts itself to sleep. However, this method cannot be used for interrupts because they do not have a process context and a different model is required. UNIX handles interrupts by allowing code to lock out all interrupts during critical sections. This must be done very carefully since blocking interrupts carries with it the potential danger that they will not be serviced in a timely manner.

When the UNIX kernel is run on multiple processors, the basic assumptions no longer apply. This is what has in the past made the UNIX model very difficult to adapt to multiple processors. The assumptions are broken from there being more than one CPU which allows code to run in parallel. It follows that interrupt handlers and processes can run at the same time, on different processors, and the non−preemption rule is no longer sufficient to ensure kernel code is not executed in parallel. This leads to what is commonly one of the most performance degrading aspects of SMP on UNIX modeled operating systems. All interrupts on all processors must be locked during critical sections of code and only one process can execute in the kernel at any time.

For further analysis of actual SMP implementation in UNIX modeled operating systems, this paper is primarily focused on one project, the volunteer FreeBSD project. The reasons for choosing this development effort over any others are simple. First, it is an openly developed project and by that virtue has more information readily available to the public. Also, because it is a mature project, it is currently in a phase of development that includes adding a next generation SMP system to the kernel. It is (primarily) developed for the i386 architecture and is derived from BSD UNIX, the version of UNIX developed at the University of California, Berkeley. This makes it an ideal candidate for examining historical SMP development on x86 architecture as well as work being done to adapt a newer, more efficient solution to the UNIX model.

FreeBSD has had SMP support since version 3.0 which was released in October of 1998. It implemented SMP using a very simple model that would simulate the single processor model in a multiple processor environment. This approach led to the creation of what is now commonly known as the Big Kernel Lock or BKL. Essentially, any process that wished to execute in the kernel must hold the BKL in order to do so. If the BKL is already held by another process, then the original process is forced to do a busy wait on the lock. This solved the problems created by breaking the non−preemptive rule but resulted in serious performance hits, in some cases , so bad that a single processor machine would execute faster (Lehey, 4). For obvious reasons, this is unacceptable.

Due to the merger of BSDi and Walnut Creek in March of 2000, the FreeBSD team was presented with a clear and provably working model for advanced SMP implementation. When the companies merged, BSD/OS source code was shared with the FreeBSD development effort in order to support their now mutual interest. Included was the new SMP implementation for BSD/OS 5.0 called SMPng ("new generation"). The importance of this is that they were given a clear path to follow toward implementation and very strong evidence that it would work well. Due to the importance imposed by the project itself of always having a working system in between releases, the migration from the old model had to be and is carefully controlled.

The key thing to accomplish first is to take the old BKL and turn it into a blocking lock instead of a spin lock. This can’t be done in the old model because interrupts lack a process context and so the BKL had to be a spin lock out of necessity. This can be accomplished by forcing interrupts to run in a process context, called an interrupt thread. There are several tradeoffs introduced by doing this. On the positive side of things, this approach allows for a uniform approach to synchronization. The benefit of uniform design is that it simplifies the operating system implementation. Unfortunately, because scheduling takes longer than interrupt overhead there is additional latency associated with handling interrupts. Also because traditionally the UNIX kernel is non−preemptive, there is a conflict with how interrupts are handled. Because they are now treated as processes, interrupts are not allowed to preempt other processes or lower priority interrupts when they are in the kernel.

To overcome the latency and scheduling issues, a technique called lazy scheduling is introduced. Lazy scheduling works as follows: "on receiving an interrupt, the interrupt stubs note the process ID of the interrupt thread, but they do not schedule the thread. Instead it continues execution in the context of the interrupted process. The thread will be scheduled only if it has to block or if the interrupt nesting level gets too deep" (Lehey, 5). Expectations are that lazy scheduling will cause negligible overhead for the majority of interrupts. Also, as it turns out, interrupt threads are not the same as regular processes. Interrupt threads never run in user space and always share the address space of process 0, the swapper. They also run at a higher priority than all user processes and have a fixed priority not related to system load. Of course, the kernel must also implement at the very least limited preemption in order for interrupt threads to function properly. Kernel preemption has already been shown to be optimal in an SMP environment by the 2.4 branch of the Linux kernel (which, as a note to the reader, is approximately one year ahead of FreeBSD in terms of SMP development).

FreeBSD has defined four basic lock types to be used with SMPng. The default locking construct is a spin/sleep mutex which is essentially a counting semaphore with a count of 1. The lock allows spinning for a set amount of time where it would be more efficient than a context switch. If the process cannot obtain the lock in sufficient time it is put in a sleep queue and woken up when the resource is available. The second type of lock is a spin mutex. These are exactly the same as a traditional spin lock and should be used only in exceptional cases when they are required. The third locking type allows for condition variables to be built on top of a mutex. They consist of a mutex and a wait queue for the condition. The fourth lock is known as a shared/exclusive lock, or sx lock and are basically read/write locks. Intuitively this allows for multiple processes to hold a read lock while only one can hold a write lock.

Now it is possible to break the BKL and slowly migrate the system towards a finer grained locking mechanism. This work was started simply by removing the BKL and implementing two new mutexes. The first mutex, Giant, is similar to the BKL except that it is a blocking mutex. This allows for parts of the system to be slowly moved out from underneath Giant. The second mutex is

sched_lock, a spin lock that protects the scheduler queues. These two locks represent the bare minimum to implement SMPng into the kernel. Migrating from out of Giant is where real performance gains can be realized and represents the majority of the work to be done in the kernel code. Because of the nature of the FreeBSD project, it is very important that an agreed upon locking strategy be in place so that the myriad of developers around the globe will have a consistent approach to locking. In typical cases "very little code locks more than one lock at a time, and when it does, it is in a very tight context. This results in relatively reliable code, but it may not result in optimum performance" (Lehey, 7). This is acceptable due to the volunteer nature of the project and it should be noted that "large UNIX vendors have found the choice of locking strategy to be a problem" (Lehey, 7) as well.

SMPng development on FreeBSD started in June of 2000 and is still undergoing active development. Real performance gains will be seen as the kernel code is migrated out from under Giant. What is most important for the 5.0 release of FreeBSD is that everything be implemented and stable. Emphasis on performance has been placed under the domain of 5.1 and later releases of the operating system. The release date of what is now termed FreeBSD−Current is somewhat uncertain at this point due to the large amount of work going into the SMP system and also towards a project called Kernel Scheduler Entities, or KSE. Although originally expected November of 2001, it has been announced that the 5.0 release will be delayed for one year or more.

Works Cited

Dillon, Matt. "FreeBSD Meeting @ USENIX." June 30, 2001: 11 pages. Online. Internet. http://people.freebsd.org/~jhb/summit/usenix01/dillon.txt .

Evans, Jason. "Pushing FreeBSD's SMP Performance to the Next Level." September 8, 2000: 1 page. Online. Internet. http://people.freebsd.org/~jasone/smp/smp_article .

Lehey, Greg. "Improving the FreeBSD SMP Implementation." May 2001: 10 pages. Online. Internet. http://www.lemis.com/~grog/SMPng/USENIX/ .

Silberschatz, Abraham; Peter Galvin, Greg Gagne, et al. *Opterating System Concepts*. 6th Edition. New York: John Wiley & Sons, Inc, 2002.