Locking in the Multithreaded FreeBSD Kernel

John H. Baldwin
The Weather Channel
jhb@FreeBSD.org, http://people.FreeBSD.org/~jhb

Abstract

About a year ago, the FreeBSD Project embarked on the ambitious task of multithreading its kernel. The primary goal of this project is to improve performance on multiprocessor (MP) systems by allowing concurrent access to the kernel while not drastically hurting performance on uniprocessor (UP) systems. As a result, the project has been dubbed the SMP next generation project, or SMPng for short.

Multithreading a BSD kernel is not just a one-time change; it changes the way that data integrity within the kernel is maintained. Thus, not only does the existing code need to be reworked, but new code must also use these different methods. The purpose of this paper is to aid kernel programmers in using these methods.

It is assumed that the audience is familiar with the data integrity methods used in the traditional BSD kernel. The paper will open with a brief overview of these traditional methods. Next, it will describe the synchronization primitives new to the multithreaded FreeBSD kernel including a set of guidelines concerning their use. Finally, the paper will describe the tools provided to assist developers in using these synchronization primitives properly.

1 Introduction

Prior to the SMPng project, SMP support in FreeBSD was limited to the i386 architecture and used one giant spin lock for the entire kernel as described in Section 10.2 of [Schimmel94]. This kernel architecture is referred to as pre-SMPng. The goal of SMPng is to allow multiple threads to execute in the kernel concurrently on SMP systems.

2 Current Status

The SMPng project first began in June of 2000 with a presentation given to several FreeBSD developers by Chuck Paterson of BSDi explaining BSDi's SMP project to multithread their own kernel. This meeting set the basic design, and developers started working on code shortly after.

The first step was to implement the synchronization primitives. Once this was done, two mutexes were added. A spin mutex named `sched_lock` was used to protect the scheduler queues, sleep queues, and other scheduler data structures. A sleep mutex named `Giant` was added to protect everything else in the kernel. The second step was to move most interrupt handlers into interrupt threads. Interrupt threads are necessary so that an interrupt handler has a context in which to block on a lock without blocking an unrelated top half kernel thread. A few interrupt handlers such as those for clock interrupts and serial I/O devices still run in the context of the thread they interrupt and thus do not have a context in which to block. Once this was done, the old `spl` synchronization primitives were no longer needed and could be converted into nops. They are still left around for reference until code is converted to use locks, however.

Now that most of the infrastructure is in place, the current efforts are directed at adding locks to various data structures so that portions of code can be taken out from under `Giant`. There are still some infrastructure changes that need to be implemented as well. These include implementing a lock profiler that can determine which locks are heavily contested as well as where they are heavily contested. This will allow developers to determine when locking needs to be made more fine-grained.

3 Problems Presented by Concurrent Access to the Kernel

Multiple threads of execution within a BSD kernel have always presented a problem. Data structures are generally manipulated in groups of operations. If two concurrent threads attempt to modify the same shared data structure, then they can corrupt that data structure. Similarly, if a thread is interrupted and another thread accesses or manipulates a data structure that the first thread was accessing or manipulating, corruption can result. Traditional BSD did not need to address the first case, so it simply had to manage the second case using the following three methods:

- Threads that are currently executing kernel code are not preempted by other threads,
- Interrupts are masked in areas of the kernel where an interrupt may access or modify data currently being accessed or modified, and
- Longer term locks are acquired by synchronizing on a lock variable via `sleep` and `wakeup`.

With the advent of MP systems, however, these methods are not sufficient to cover both problematic cases. Not allowing threads in the kernel to be preempted does nothing to prevent two threads on different CPUs from accessing the same shared data structure concurrently. Interrupt masking only affects the current CPU, thus an interrupt on one CPU could corrupt data structures being used on another CPU. The third method is not completely broken since the locks are sufficient to protect the data they protected originally. However, race conditions on the locking and unlocking thread itself can lead to temporary hangs of threads. For a more detailed explanation see Chapter 8 of [Schimmel94] and Section 7.2 of [Vahalia96]. For an explanation of how these protections were implemented in 4.4BSD and derivatives see [McKusick96].

The pre-SMPng kernel addressed this situation on SMP systems by only allowing one processor to execute in the kernel at a time. This preserved the UP model for the kernel at the expense of disallowing any concurrent access to the kernel.

4 Basic Tools

Fortunately, MP-capable CPUs provide two mechanisms to deal with these problems: atomic operations and memory barriers.

4.1 Atomic Operations

An atomic operation is any operation that a CPU can perform such that all results will be made visible to each CPU at the same time and whose operation is safe from interference by other CPUs. For example, reading or writing a word of memory is an atomic operation. Unfortunately, reading and writing are only of limited usefulness alone as atomic operations. The most useful atomic operations allow modifying a value by both reading the value, modifying it, and writing it as a single atomic change. The details of FreeBSD's atomic operation API can be found in the atomic manual page [Atomic]. A more detailed explanation of how atomic operations work can be found in Section 8.3 of [Schimmel94].

Atomic operations alone are not very useful. An atomic operation can only modify one variable. If one needs to read a variable and then make a decision based on the value of that variable, the value may change after the read, thus rendering the decision invalid. For this reason, atomic operations are best used as building blocks for higher level synchronization primitives or for noncritical statistics.

4.2 Memory Barriers

Many modern CPUs include the ability to reorder instruction streams to increase performance [Intel00, Schimmel94, Mauro01]. On a UP machine, the CPU still operates correctly so long as dependencies are satisfied by either extra logic on the CPU or hints in the instruction stream. On a SMP machine, other CPUs may be operating under different dependencies, thus the data they see may be incorrect. The solution is to use memory barriers to control the order in which memory is accessed. This can be used to establish a common set of dependencies among all CPUs. An explanation of using store barriers in unlock operations can be found in Section 13.5 of [Schimmel94].

In FreeBSD, memory barriers are provided via the atomic operations API. The API is modeled on the memory barriers provided on the IA64 CPU which are described in Section 4.4.7 of [Intel00]. The API include two types of barriers: acquire and release. An acquire barrier guarantees that the current atomic operation will complete before any following memory operations. This type of barrier is used when acquiring a lock to guarantee that the lock is acquired before any protected operations are performed. A release barrier guarantees that all preceding memory operations will be completed and the results visible before the current atomic operation completes. As a result, all protected operations will only occur while the lock is held. This allows a dependency to be established between a lock and the data it protects.

5 Synchronization Primitives

Several synchronization primitives have been introduced to aid in multithreading the kernel. These primitives are implemented by atomic operations and use appropriate memory barriers so that users of these primitives do not have to worry about doing it themselves. The primitives are very similar to those used in other operating systems including mutexes, condition variables, shared/exclusive locks, and semaphores.

5.1 Mutexes

The mutex primitive provides mutual exclusion for one or more data objects. Two versions of the mutex primitive are provided: spin mutexes and sleep mutexes.

Spin mutexes are a simple spin lock. If the lock is held by another thread when a thread tries to acquire it, the second thread will spin waiting for the lock to be released. Due to this spinning nature, a context switch cannot be performed while holding a spin mutex to avoid deadlocking in the case of a thread owning a spin lock not being executed on a CPU and all other CPUs spinning on that lock. An exception to this is the scheduler lock, which must be held during a context switch. As a special case, the ownership of the scheduler lock is passed from the thread being switched out to the thread being switched in to satisfy this requirement while still protecting the scheduler data structures. Since the bottom half code that schedules threaded interrupts and runs non-threaded interrupt handlers also uses spin mutexes, spin mutexes must disable interrupts while they are held to prevent bottom half code from deadlocking against the top half code it is interrupting on the current CPU. Disabling interrupts while holding a spin lock has the unfortunate side effect of increasing interrupt latency.

To work around this, a second mutex primitive is provided that performs a context switch when a thread blocks on a mutex. This second type of mutex is dubbed a sleep mutex. Since a thread that contests on a sleep mutex blocks instead of spinning, it is not susceptible to the first type of deadlock with spin locks. Sleep mutexes cannot be used in bottom half code, so they do not need to disable interrupts while they are held to avoid the second type of deadlock with spin locks.

As with Solaris, when a thread blocks on a sleep mutex, it propagates its priority to the lock owner. Therefore, if a thread blocks on a sleep mutex and its priority is higher than the thread that currently owns the sleep mutex, the current owner will inherit the priority of the first thread. If the owner of the sleep mutex is blocked on another mutex, then the entire chain of threads will be traversed bumping the priority of any threads if needed until a runnable thread is found. This is to deal with the problem of priority inversion where a lower priority thread blocks a higher priority thread. By bumping the priority of the lower priority thread until it releases the lock the higher priority thread is blocked on, the kernel guarantees that the higher priority thread will get to run as soon as its priority allows.

These two types of mutexes are similar to the Solaris spin and adaptive mutexes. One difference from the Solaris API is that acquiring and releasing a spin mutex uses different functions than acquiring and releasing a sleep mutex. A difference with the Solaris implementation is that sleep mutexes are not adaptive. Details of the Solaris mutex API and implementation can be found in section 3.5 of [Mauro01].

5.2 Condition Variables

Condition variables provide a logical abstraction for blocking a thread while waiting for a condition.

Condition variables do not contain the actual condition to test, instead, one locks the appropriate mutex, tests the condition, and then blocks on the condition variable if the condition is not true. To prevent lost wakeups, the mutex is passed in as an interlock when waiting on a condition.

FreeBSD's condition variables use an API quite similar to those provided in Solaris. The only differences being the lack of a `cv_wait_sig_swap` and the addition of `cv_init` and `cv_destroy` constructors and destructors. The implementation also differs from Solaris in that the sleep queue is embedded in the condition variable itself instead of coming from the hashed pool of sleep queue's used by `sleep` and `wakeup`.

5.3 Shared/Exclusive Locks

Shared/Exclusive locks, also known as sx locks, provide simple reader/writer locks. As the name suggests, multiple threads may hold a shared lock simultaneously, but only one thread may hold an exclusive lock. Also, if one thread holds an exclusive lock, no threads may hold a shared lock.

FreeBSD's sx locks have some limitations not present in other reader/writer lock implementations. First, a thread may not recursively acquire an exclusive lock. Secondly, sx locks do not implement any sort of priority propagation. Finally, although upgrades and downgrades of locks are implemented, they may not block. Instead, if an upgrade cannot succeed, it returns failure, and the programmer is required to explicitly drop its shared lock and acquire an exclusive lock. This design was intentional to prevent programmers from making false assumptions about a blocking upgrade function. Specifically, a blocking upgrade must potentially release its shared lock. Also, another thread may obtain an exclusive lock before a thread trying to perform an upgrade. For example, if two threads are performing an upgrade on a lock at the same time.

5.4 Semaphores

FreeBSD's semaphores are simple counting semaphores that use an API similar to that of POSIX.4 semaphores [Gallmeister95]. Since `sema_wait` and `sema_timedwait` can potentially block, mutexes must not be held when these functions are called.

6 Guidelines

Simply knowing how a lock functions or what API it uses is not sufficient to understand how a lock should be used. To help guide kernel developers in using locks properly, several guidelines have been crafted. Some of these guidelines were provided by the BSD/OS developers while others are the results of the experiences of FreeBSD developers.

6.1 Special Rules About `Giant`

The `Giant` sleep mutex is a temporary mutex used to protect data structures in the kernel that are not fully protected by another lock during the SMPng transition. `Giant` has to not interfere with other locks that are added. Thus, `Giant` has some unique properties.

First, no lock is allowed to be held while acquiring `Giant`. This ensures that other locks can always be safely acquired whether or not `Giant` is held. This in turn allows subsystems that have their own locks to be called directly without `Giant` being held, and to be called by other subsystems that still require `Giant`.

Second, the `Giant` mutex is automatically released by the `sleep` and condition variable wait function families before blocking and reacquired when a thread resumes. Since the `Giant` mutex is reacquired before the interlock lock and no other mutexes may be held while blocked, this does not result in any lock order reversals due to the first property. There are lock order reversals between the `Giant` mutex and locks held while a thread is blocked when the thread is resumed, but these reversals are not problematic since the `Giant` mutex is dropped when blocking on such locks.

6.2 Avoid Recursing on Exclusive Locks

A lock acquire is recursive if the thread trying to acquire the lock already holds the lock. If the lock is recursive, then the acquire will succeed. A recursed lock must be released by the owning thread the same number of times it has been acquired before it is fully released.

When an exclusive lock is acquired, the holder usually assumes that it has exclusive access to the object the lock protects. Unfortunately, recursive locks can break this assumption in some cases. Suppose we have a function F1 that uses a recursive lock L to protect object O. If function F2 acquires lock L, modifies object O so that is in an inconsistent state, and calls F1, F1 will recursively acquire L1 and falsely assume that O is in a consistent state.

One way of preventing this bug from going undetected is to use a non-recursive lock. Lock assertions can be placed in internal functions to ensure that calling functions obtain the necessary locks. This allows one to develop a locking protocol whereby callers of certain functions must obtain locks before calling the functions This must be balanced, however, with a desire to not require users of a public interface to acquire locks private to the subsystem.

For example, suppose a certain subsystem has a function G that is a public function called from other functions outside of this subsystem. Suppose that G is also called by other functions within the subsystem. Now there is a tradeoff involved. If you use assertions to require a lock to be held when G is called, then functions from other subsystems have to be aware of the lock in question. If, on the other hand, you allow recursion to make the interface to the subsystem cleaner, you can potentially allow the problem described above.

A compromise is to use a recursive lock, but to limit the places where recursion is allowed. This can be done by asserting that an acquired lock is not recursed just after acquiring it in places where recursion is not needed. However, if a lock is widely used, then it may be difficult to ensure that all places that acquire the lock make the proper assertions and that all places that the lock may be recursively acquired are safe.

An example of the first method is used with the `psignal` function. This function posts a signal to a process. The process has to be locked by the per-process lock during this operation. Since `psignal` is called from several places even including a few device drivers, it was desirable to acquire the lock in `psignal` itself. However, in other places such as signaling a parent process with `SIGCHLD` during process exit, several operations need to be performed while holding the process lock. This resulted in recursing on the process lock. Due to the wide use of the process lock, it was determined that the lock should remain non-recursive. Thus, `psignal` asserts that a process being signaled is locked, and callers are required to lock the process explicitly.

Currently in FreeBSD, mutexes are not recursive by default. A mutex can be made recursive by passing a flag to `mtx_init`. Exclusive sx locks never allow recursion, but shared sx locks always allow recursion. If a thread attempts to recursively lock a non-recursive lock, the kernel will panic reporting the file and line number of the offending lock operation.

6.3 Avoid Holding Exclusive Locks for Long Periods of Time

Exclusive locks reduce concurrency and should be held for as short a time as possible. Since a thread can block for an indeterminate amount of time, it follows that exclusive locks should not be held by a blocked thread when possible. Locks that should be held by a blocked thread are protecting data structures already protected in the pre-SMPng kernel. Thus, only locks present prior to SMPng should be held by a blocked thread, but new locks should not be held while blocked. The existing locks should be sufficient to cover the cases when an object needs to be locked by a blocked thread. An example of this type of lock would be a lock associated with a file's contents.

Functions that block such as the `sleep` and `cv_wait` families of functions should not be called with any locks held. Exceptions to this rule are the `Giant` lock, the optional interlock lock passed to the function, and locks that can be held by a blocked thread. Since these functions only release the interlock once, they should not be called with the interlock recursively acquired.

Note that it is better to hold a lock for a short period of time when it is not needed than to drop the lock only to reacquire it. Otherwise, the thread may have to block when it tries to reacquire the lock. For example, suppose lock L protects two lists A and B and that the objects on the lists do not need locks since they only have one reference at a time. Then, a section of code may want to remove an object from list A, set a flag in the object, and store the object on list B. The operation of setting the flag does not require a lock due to the nature of the object, thus the code could drop the lock around that operation. However, since the operations to drop and acquire the lock are more expensive than the operation to set a flag, it is better to lock L, remove an object from list A, set the flag in the object, put the object on list B, and then release the lock.

6.4 Use Sleep Mutexes Rather Than Spin Mutexes

As described earlier, spin mutexes block interrupts while they are held. Thus, holding spin mutexes can increase interrupt latency and should be avoided when possible. Sleep mutexes require a context in case they block, however, so sleep mutexes may not be used in interrupt handlers that do not run in a thread context. Instead, these handlers must use spin mutexes. Spin mutexes may also need to be used in non-interrupt contexts or in threaded interrupt handlers to protect data structures shared with non-threaded interrupt handlers. All other mutexes should be sleep mutexes to avoid increasing interrupt latency. Also note that since locking a sleep mutex may potentially block, a sleep mutex may not be acquired while holding a spin mutex. Unlocking a contested sleep mutex may result in switching to a higher priority thread that was waiting on the mutex. Since we cannot switch while holding a spin mutex, this potential switch must be disabled by passing the `MTX_NOSWITCH` flag when releasing a sleep mutex while holding a spin mutex.

6.5 Lock Both Reads and Writes

Data objects protected by locks must be protected for both reads and writes. Specifically, if a data object needs to be locked when writing to the object, it also needs to be locked when reading from the object. However, a read lock does not have to be as strong as a write lock. A read lock simply needs to ensure that the data it is reading is coherent and up to date. Memory barriers within the locks guarantee that the data is not stale. All that remains for a read lock is that the lock block all writers until it is released. A write lock, on the other hand, must block all readers in addition to meeting the requirements of a read lock. Note that a write lock is always sufficient protection for reading.

Read locks and write locks can be implemented in several ways. If an object is protected by a mutex, then the mutex must be held for read and write locks. If an object is protected by an sx lock, then a shared lock is sufficient for reading, but an exclusive lock is required for writing. If an object is protected by multiple locks, then a read lock of at least one of the locks is sufficient for reading, but a write lock of all locks is required for writing.

An example of this method is the pointer to the parent process within a process structure. This pointer is protected by both the `proctree_lock` sx lock and the per-process mutex. Thus, to read the parent process pointer, one only needs to either grab a shared or exclusive lock of the `proctree_lock` or lock the process structure itself. However, to set the parent process pointer, both locks must be locked exclusively. Thus, the `inferior` function asserts that the `proctree_lock` is locked while it walks up the process tree seeing if a specified process is a child of the current process, while `psignal` simply needs to lock the child process to read the parent process pointer while posting `SIGCHLD` to the parent. However, when re-parenting a process, both the child process and the process tree must be exclusively locked.

7 Diagnostic Tools

FreeBSD provides two tools that can be used to ensure correct usage of locks. The tools are lock assertions and a lock order verifier named witness. To demonstrate these tools, we'll provide code examples from a kernel module [Crash] and then explain how the tools can be used. The module works by receiving events from a userland process via a sysctl. The event is then handed off to a kernel thread which performs the tasks associated with a specific event.

Both of these tools are only enabled if the kernel is compiled with support for them enabled. This allows a kernel tuned for performance to avoid the overhead of verifying the assertions while allowing for a debug kernel used in development to perform the extra checks. Currently both of these tools are enabled by default, but they will be disabled when the FreeBSD Project releases 5.0 for performance reasons. If either tool detects a fatal problem, then it will panic the kernel. If the kernel debugger is compiled in, then the panic will drop into the kernel debugger as well. For non-fatal problems, the tool may drop into the kernel debugger if the debugger is enabled and the tool is configured to do so.

7.1 Lock Assertions

Both mutexes and shared/exclusive locks provide macros to assert that a lock is held at any given point. For mutexes, one can assert that the current thread either owns the lock or does not own the lock. If the thread does own the lock, one can also assert that the lock is either recursed or not recursed. For shared/exclusive locks, one can assert that a lock is either locked in either fashion. If an assertion fails, then the kernel will panic and drop into the kernel debugger if the debugger is enabled.

For example, in the crash kernel module, events 14 and 15 trigger false lock assertions. Event 14 asserts that `Giant` is owned when it is not.

```
        case 14:
                mtx_assert(&Giant, MA_OWNED);
                break;
```

When event 14 is triggered, the output on the kernel console is:

```
crash: assert that Giant is locked
panic: mutex Giant not owned at crash.c:202
cpuid = 3; lapic.id = 03000000
Debugger("panic")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

Event 15 exclusively locks the sx lock `foo` and then asserts that it is share locked.

```
        case 15:
                sx_xlock(&foo);
                sx_assert(&foo, SX_SLOCKED);
                sx_xunlock(&foo);
```

When event 15 is triggered, the output on the kernel console is:

```
crash: assert that foo is slocked
 while it is xlocked
panic: Lock (sx) foo exclusively locked
 @ crash.c:206.
cpuid = 1; lapic = 01000000
Debugger("panic")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

7.2 Witness

Along with the mutex code and advice provided by BSDi came a lock order verifier called witness. A lock order verifier checks the order in which locks are acquired against a specified lock order. If locks are acquired out of order, than the code in question may deadlock against other code which acquires locks in the proper order. For the purposes of lock order, a lock A is said to be acquired before lock B, if lock B is acquired while holding lock A.

Witness does not use a static lock order, instead it dynamically builds a tree of lock order relationships. It starts with explicit lock orders hard-coded in the source code. Once the system is up and running, witness monitors lock acquires and releases to dynamically add new order relationships and report violations of previously established lock orders. For example, if a thread acquires lock B while holding lock A, then witness will save the lock order relationship of A before B in its internal state. Later on if another thread attempts to acquire lock B while holding lock A, it will print a warning message on the console and optionally drop into the kernel debugger. The BSD/OS implementation only worked with mutexes, but the FreeBSD Project has extended it to work with shared/exclusive locks as well.

Locks are grouped into classes based on their names. Thus, witness will treat two different locks with the same name as the same. If two locks with the same name are acquired, witness will panic unless multiple acquires of locks of that name are explicitly allowed. This is because witness can not check the order in which locks of the same name are acquired. The only lock group that witness currently allows multiple acquires of member locks are process locks. This is safe because process locks follow a defined order of locking a child process before locking a parent process.

The crash kernel module was originally written to test witness on sx locks, thus it contains events to violate lock orders to ensure that witness detects the reversals. Event 2 locks the sx lock `foo` followed by the the sx lock `bar`. Event 3 locks the sx lock `bar` followed by the sx lock `foo`.

```
        case 2:
                sx_slock(&foo);
                sx_slock(&bar);
                sx_sunlock(&foo);
                sx_sunlock(&bar);
                break;
        case 3:
                sx_slock(&bar);
                sx_slock(&foo);
                sx_sunlock(&foo);
                sx_sunlock(&bar);
                break;
```

When event 2 is triggered followed by event 3, the output on the kernel console is:

```
crash: foo then bar
crash: bar then foo
lock order reversal
 1st 0xc335fce0 bar @ crash.c:142
 2nd 0xc335fc80 foo @ crash.c:143
Debugger("witness_lock")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

Event 4 locks the sx lock `bar`, locks the sx lock `foo`, and then locks the sx lock `bar2`. The locks `bar` and `bar2` both have the same name and thus belong to the same lock group.

```
        case 4:
                sx_slock(&bar);
                sx_slock(&foo);
                sx_slock(&bar2);
                sx_sunlock(&bar2);
                sx_sunlock(&foo);
                sx_sunlock(&bar);
                break;
```

When event 4 is triggered, the output on the kernel console is:

```
crash: bar then foo then bar
lock order reversa# l
 1st 0xc3363ce0 bar @ crash.c:148
 2nd 0xc3363c80 foo @ crash.c:149
 3rd 0xc3363d40 bar @ crash.c:150
Debugger("witness_lock")
Stopped at      Debugger+0x46:  pushl   %ebx
db>
```

Note that if there are actually three locks involved in a reversal, all three are displayed. However, if two locks are merely reversing a previously established order, only the information about the two locks is displayed.

In addition to performing checks, witness also adds a new command to the kernel debugger. The command `show locks [pid]` displays the locks held by a given thread. If a pid is not specified, then the locks held by the current thread are displayed. For example, after the panic in the previous example, the output is:

```
db> show locks
shared (sx) foo (0xc3363c80) locked
 @ crash.c:149
shared (sx) bar (0xc3363ce0) locked
 @ crash.c:148
```

Sleep mutexes and sx locks are displayed in the order they were acquired. If the thread is currently executing on a CPU, then any spin locks held by the current thread are displayed as well.

8 Conclusion

Multithreading a BSD kernel is not a trivial task. However, it is a feasible task given the proper tools and some guidelines to avoid the larger pitfalls. Volunteers seeking to work on the SMPng Project can check the SMPng Project web page [SMPng] for the todo list. The FreeBSD SMP mailing list (`freebsd-smp@FreeBSD.org`) is also available for those wishing to discuss issues with other developers.

9 Acknowledgments

Thanks to The Weather Channel; Wind River Systems, Inc.; and Berkeley Software Design, Inc. for funding some of the SMPng development as well as this paper. Thanks also to BSDi for donating their SMP code from the BSD/OS 5.0 branch. Special thanks are due to those who helped review and critique this paper including Sam Leffler, Robert Watson, and Peter Wemm. Finally, thanks to all the contributors to the SMPng Project including Jake Burkholder, Matt Dillon, Tor Egge, Jason Evans, Brian Feldman, Andrew Gallatin, Greg Lehey, Jonathan Lemon, Bosko Milekic, Mark Murray, Chuck Paterson, Alfred Perlstein, Doug Rabson, Robert Watson, Andrew Reiter, Dag-Erling Smørgav, Seigo Tanimura, and Peter Wemm.

10 Availability

FreeBSD is an Open Source operating system licensed under the very liberal Berkeley-style license [FreeBSD]. It is freely available from a world-wide network of FTP servers. The SMPng work is being done in the development branch and is not presently in any released version of FreeBSD. It will debut in FreeBSD 5.0. Installation snapshots for the development versions of FreeBSD on the i386 architecture can be found at

```
ftp://snapshots.jp.FreeBSD.org/ \
   pub/FreeBSD/snapshots/i386
```

References

[Gallmeister95] Bill Gallmeister, *POSIX.4 Programming for the Real World*, O'Reilly & Associates, Inc. (1995) p. 134-146.

[Intel00] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 1: IA-64 Application Architecture*, Intel Corporation (2000).

[Mauro01] Jim Mauro and Richard McDougall, *Solaris Internals: Core Kernel Components*, Sun Microsystems Press (2001).

[McKusick96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Longman, Inc. (1996) p. 91-92.

[Schimmel94] Curt Schimmel, *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*, Addison-Wesley Publishing Company (1994).

[Vahalia96] Uresh Vahalia, *UNIX Internals: The New Frontiers*, Prentice-Hall, Inc. (1996).

[Atomic] *Atomic*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/ \ man.cgi?manpath=FreeBSD+5.0-current`

[Crash] Crash Example Kernel Module, `http://people.FreeBSD.org/~jhb/ \ crash/crash.c`

[FreeBSD] FreeBSD Project, `http://www.FreeBSD.org`

[SMPng] FreeBSD SMPng Project, `http://www.FreeBSD.org/smp`