Improving the FreeBSD SMP implementation
A case study

Greg Lehey
The FreeBSD Project
grog@FreeBSD.org

30 October 2003

ABSTRACT

UNIX-derived operating systems have traditionally have a simplistic approach to process synchronization which is unsuited to multiprocessor application. Initial FreeBSD SMP support kept this approach by allowing only one process to run in kernel mode at any time, and also blocked interrupts across multiple processors, causing seriously suboptimal performance of I/O bound systems. This paper describes the work done to remove this bottleneck, paying particular attention to the project management aspects and the particular challenges of a large open source development project.

Introduction

Without doubt one of the most remarkable phenomena in the history of computers is the emergence of the Free Software movement, often called "Open Source". Undisciplined groups of programmers with above-average ability and extreme motivation have set out to produce better software than established commercial companies. Their success has been so great that many of the established companies have taken on the cause and are running their own free software projects.

The press has taken up the cause in its usual exaggerated way: Open Source is the way of the future, commercial software is doomed, and of course many people, both inside the industry and outside. The truth is more differentiated, of course. This paper looks at some of the issues in a real-life development project.

The problem

An interesting aspect of free UNIX and UNIX-like implementations such as FreeBSD and Linux is that their performance and stability on machines of the early 90's was often significantly better than the performance of commercial platforms, notably Microsoft. Comparisons have shown that the difference in performance might appear to be several orders of magnitude. For example, a few years ago Microsoft recommended three Compaq ProLiant 5000s or 5500s with four Pentium Pro processors and 512 MB memory each in order to serve 6 GB a day via *ftp*. At any one time, only one of the systems was active; the others were needed for failover if the active system crashed. At the same time, *wcarchive.cdrom.com*, a single Pentium Pro running FreeBSD, was delivering 700 GB per day. It got by just fine without failover.

The free OS community had become so used to this state of affairs that it came as a shock when in April 1999 Microsoft published the results of a benchmark done by Mindcraft, which showed that Microsoft NT outperformed Linux by a factor of up to three times.

Parts of the Linux community reacted swiftly: they claimed that the benchmark results were wrong, that Mindcraft deliberately didn't tune the Linux installations they used, and a number of other things. Others in the Linux community looked more carefully and observed that, though the benchmarks measured a rather contrived use of web servers, they did in fact show an area where Microsoft significantly outperformed Linux.

More careful analysis of the tests showed that one of the chief problems with the Linux approach was the poor multiprocessor (SMP) support.

FreeBSD has a reputation for significantly outperforming Linux, so it would be reasonable for the FreeBSD community to repeat the tests. Instead, the FreeBSD community kept very quiet: FreeBSD had exactly the same problem with the SMP implementation. The following sections summarize the issues.

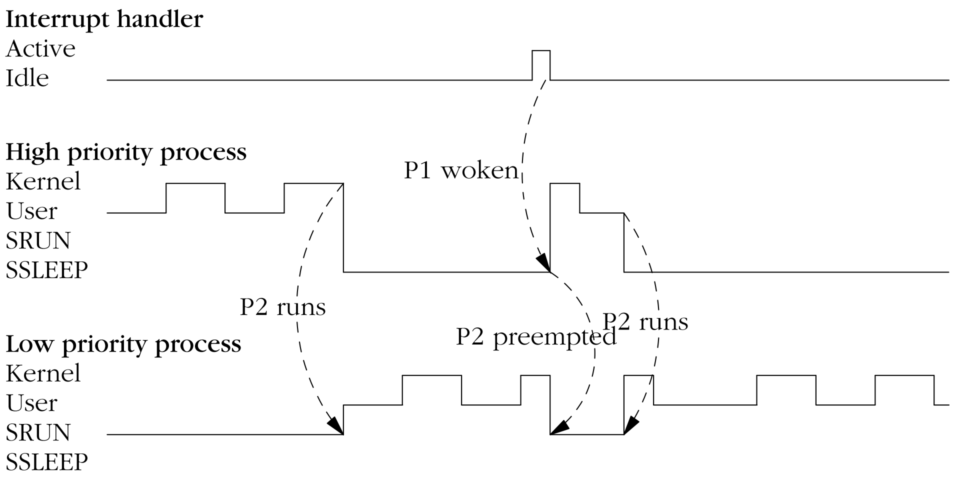
The old FreeBSD SMP implementation

A crucial issue in the design of an operating system is the manner in which it shares resources such as memory, data structures and processor time. In the UNIX model, the main clients for resources are processes and interrupt handlers. Interrupt handlers operate completely in kernel space, primarily on behalf of the system. Processes normally run in one of two different modes, user mode and kernel mode. User mode code is the code of the program from which the process is derived, and kernel mode code is part of the kernel. This structure gives rise to multiple potential conflicts. The most obvious demand a process or interrupt routine places on the system is that it wants to run: it must execute instructions. In traditional UNIX, the rules governing this sharing are:

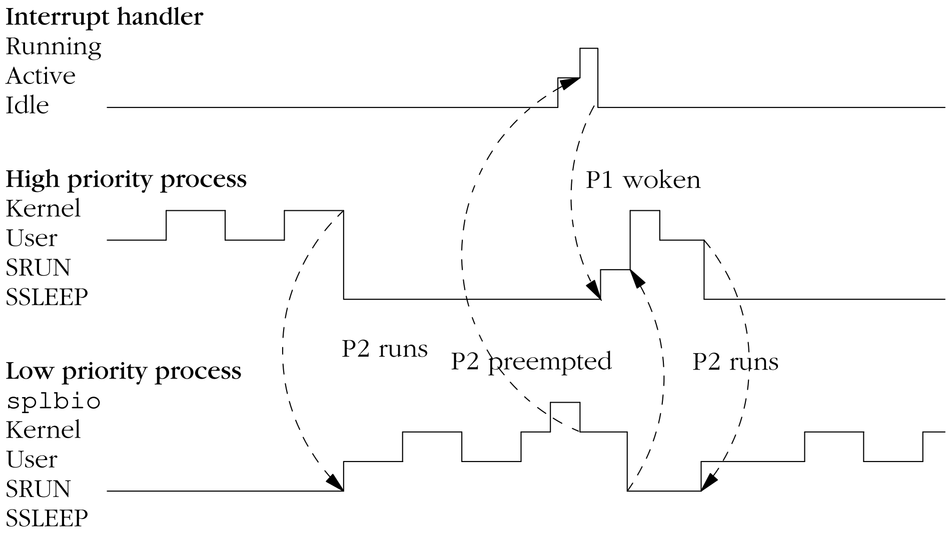
- There is only one processor. All code runs on it.
- If both an interrupt handler and a process are available to run, the interrupt handler runs.
- Interrupt handlers have different priorities. If one interrupt handler is running and one with a higher priority becomes runnable, the higher priority interrupt immediately preempts the lower priority interrupt.

- The scheduler runs when a process voluntarily relinquishes the processor, its time slice expires, or a higher-priority process becomes runnable. The scheduler chooses the highest priority process which is ready to run.
- If the process is in kernel mode when its time slice expires or a higher priority process becomes runnable, the system waits until it returns to user mode or sleeps before running the scheduler.

The following diagrams illustrate the difference caused by waiting for the process to relinquish the kernel:



Ideal single processor scheduling



Real single processor scheduling

As can be seen, the only difference is in the slight delay before the high-priority process regains control. The processor always has something constructive to do, so the overall throughput remains unchanged. This method works acceptably for the single processor machines for which it was designed. In the following section, we'll see the reasoning behind the last decision.

Kernel data objects

The most obvious problem is access to memory. Modern UNIX systems run with *memory protection*, which prevents processes in user mode from accessing the address space of other processes. This protection no longer applies in kernel mode: all processes share the kernel address space, and they need to access data shared between all processes. For example, the `fork()` system call needs to allocate a `proc` structure for the new process by taking the first available `struct proc` entry off a freelist and initializing it.

Clearly it is necessary to ensure that the currently executing process is not interrupted while manipulating this list, maybe because a higher priority process wants to run. If it is interrupted while it is removing the entry from the list, and the interrupting code also allocates a process structure, it is possible for both processes to use the same `struct proc` entry, creating the process equivalent of Siamese twins.

UNIX solves this issue with the rule "The UNIX kernel is non-preemptive". This means that when a process is running in kernel mode, no other process can execute kernel code until the first process relinquishes the kernel voluntarily, either by returning to user mode, or by sleeping.

Synchronizing processes and interrupts

The non-preemption rule only applies to processes. Interrupts happen independently of process context, so a different method is needed. In device drivers, the process context ("top half") and the interrupt context ("bottom half") must share data. Two separate issues arise here: each half must ensure that any changes to shared data structures occur in a consistent manner, and they must find a way to synchronize with each other.

Protection

Interrupt handlers cannot be interrupted by processes, and they don't normally share data structures with other interrupt handlers, so they don't need to perform anything in particular to protect themselves. On the other hand, the process context must take steps to ensure that it is not interrupted while manipulating shared data. They do this with one of the `spl`*x* functions, which lock out specific interrupts. One of the most significant sources of bugs in drivers is inadequate synchronization with the bottom half.

Waiting for the other half

In other cases, a process will need to wait for some event to complete. The most obvious example is I/O: a process issues an I/O request, and the driver initiates the transfer. It can be a long time before the transfer completes: if it's reading keyboard input, for example, it could be weeks before the I/O completes. When the transfer completes, it causes an interrupt, so it's the interrupt handler which finally determines that the transfer is complete and notifies the process. Traditional UNIX performs this synchronization with the functions `sleep` and `wakeup`. The top half of a driver calls `sleep` when it wants to wait for an event, and the bottom half calls `wakeup` when the event occurs.

It's important to note that this model is asymmetric: processes perform synchronization, interrupt handlers don't.

Adapting the UNIX model to SMP

A number of the basic assumptions of this model no longer apply to SMP, and others become more of a problem:

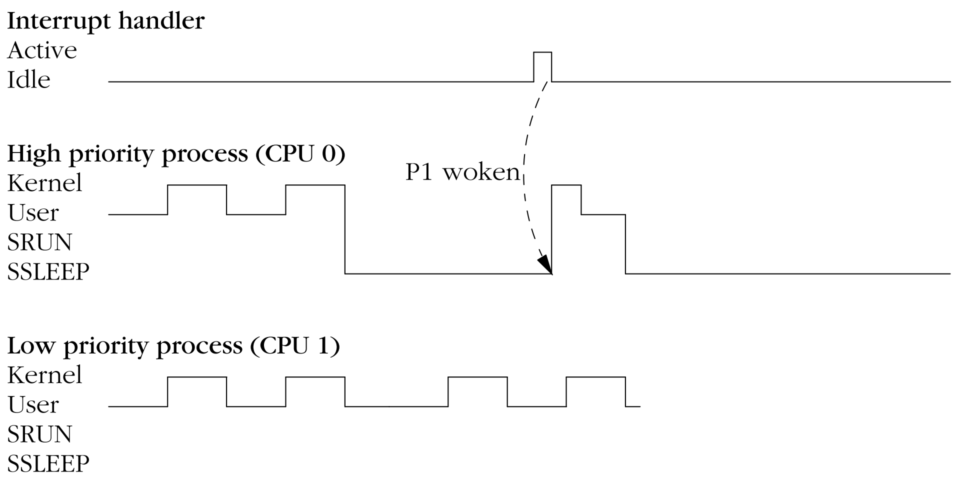
- More than one processor is available. Processes can run in parallel.
- Interrupt handlers and user processes can run on different processors at the same time.
- The "non-preemption" rule is no longer sufficient to ensure that two processes can't execute at the same time, so it would theoretically be possible for two processes to allocate the same memory.
- Locking out interrupts must happen in every processor. This can adversely affect performance.

One of the most far-reaching aspects of this situation is that interrupt handlers can no longer rely on having the system to themselves. They must find a way of synchronizing, something they were never intended to do. Interrupt lockout is no longer sufficient, and the only alternative is some variant on the `sleep/wakeup` paradigm. But interrupt handlers aren't equipped to sleep.

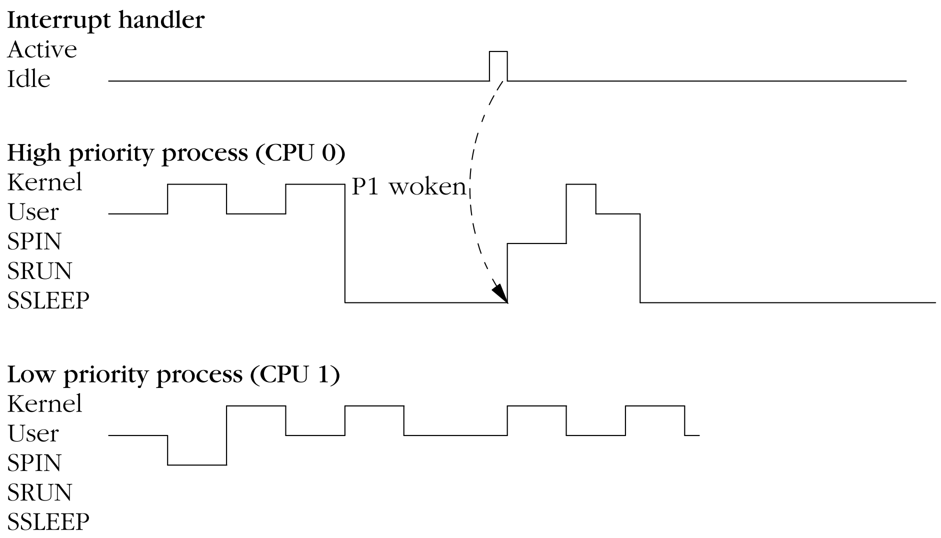
The initial FreeBSD model

The original version of FreeBSD SMP support solved these problems in a manner designed for reliability rather than performance: effectively it found a method to simulate the single-processor paradigm on multiple processors. Specifically, only one process could run in the kernel at any one time. The system ensured this with a spinlock, the so-called *Big Kernel Lock* (*BKL*), which ensured that only one processor could be in the kernel at a time. On entry to the kernel, each processor attempted to get the BKL. If another processor was executing in kernel mode, the other processor performed a *busy wait* until the lock became free. In an extreme case, this waiting could degrade SMP performance to below that of a single processor machine.

The following diagrams show the effect of the BKL:

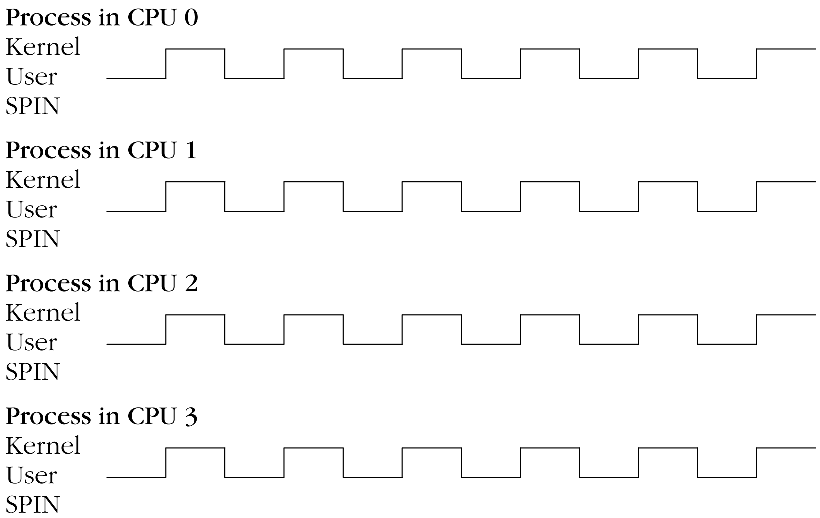


Ideal dual processor scheduling

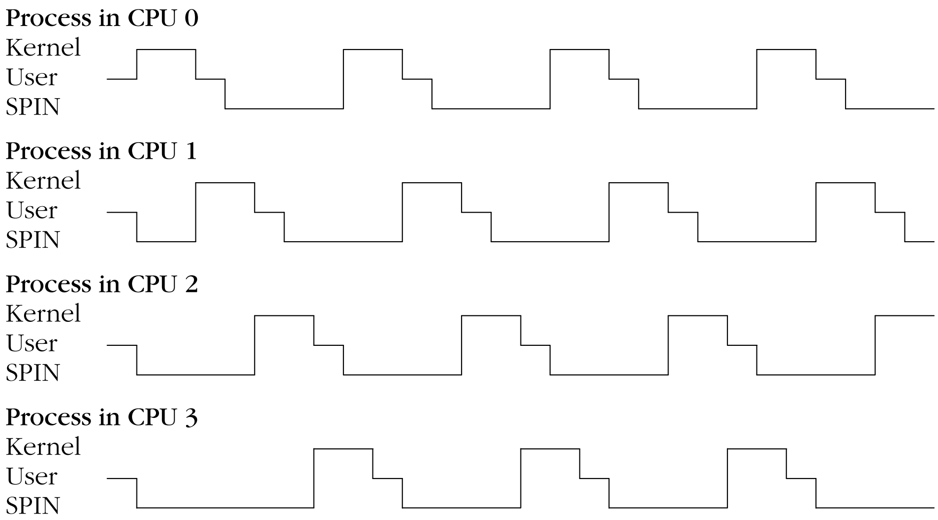


Real dual processor scheduling

In this example, the low priority process must spin while waiting to enter the kernel. Also, after being woken the high priority process spins until the low priority process leaves the kernel. This causes a significant waste of time. Things are even more extreme on a four processor system with four processes running without sleeping:



Ideal quad processor scheduling



Real quad processor scheduling

In this example, each processor spends more than 50% of its time spinning. The overall throughput is less than that of an ideal two-processor system.

Improving the SMP implementation

Multiple processor machines have been around for a long time, since before UNIX was written. During this time, a number of solutions to this kind of problem have been devised. The problem was less to find a solution than to find a solution which would fit in the UNIX environment, and which was simple enough to be implemented by a group of volunteers. The Linux community, with some commercial backing, went their own way, coming up with a pragmatic result which addressed the issues at hand, but which did not satisfy many of the members of the FreeBSD project. Instead, for quite some time, FreeBSD did nothing.

Part of the problem was the nature of the project: we weren't selling anything, we were working for our personal satisfaction. If nobody who suffered as a result of the performance issues was prepared to do the work, it wouldn't happen. In addition, any changes would be pervasive, and they would require both a lot of work and cooperation on an unprecedented scale.

Salvation

In early 2000, a solution presented itself. The FreeBSD project had been largely supported by Walnut Creek CDROM. With the advent of high-speed Internet links, the CD-ROM market was no longer what it used to be, and Walnut Creek merged with BSDi, the vendors of the commercial BSD variant BSD/OS. Initially there was much talk of merging the operating systems as well, but this met with considerable resistance in some areas. Finally a compromise was reached: FreeBSD would gain access to the complete BSD/OS source code and would be free to merge any reasonable part of the operating system.

BSDi already had a functional prototype of greatly improved SMP support, dubbed SMPng. The FreeBSD project decided to import the code into FreeBSD and also adopted the name. Many of the BSDi engineers came from Sun Microsystems, and they brought some of the Solaris ideas with them. One of the most significant of these ideas was the concept of interrupt threads, which enabled interrupt handlers to block. This enabled removal of the giant lock in favour of finer-grained locking.

The initial meeting

In June 2000, the FreeBSD project held a two-day meeting at the Yahoo! complex in Sunnyvale CA in order to determine how to proceed with importing the code. This was just before the USENIX conference down the road in Monterey, and a total of 20 people attended. There was no restriction on attendance: all people who wished to attend did so. Three attendees were from Apple Computer, two from BSDi, three from Yahoo!, and eleven from the FreeBSD project. Only about half these people had kernel development experience.

The meeting started with a discussion of the current BSDi code and the issues at hand, presented by Chuck Paterson, BSDi's chief developer.

SMP, new generation

BSDi's new implementation of SMP was never released: BSDi was acquired by Wind River Systems, who stopped further development in early 2003 and who have since ceased marketing the product. The following discussion describes both the initial BSDi version and the changes made during the implementation of FreeBSD SMPng.

The most radical difference in SMPng are:

- Interrupt code ("bottom half") now runs in a process context, enabling it to block if necessary. This process context is termed an *interrupt thread.*
- Interrupt lockout primitives (`spl`*foo*) have been removed. The low-level interrupt code still needs to block interrupts briefly, but the interrupt service routines themselves run with interrupts enabled. Instead of locking out interrupts, the system uses mutexes, which may be either spin locks or blocking locks.

Interrupt threads

The single most important aspect of the implementation is the introduction of a process or "thread" context for interrupt handlers. This change involves a number of tradeoffs:

- The process context allows a uniform approach to synchronization: it is no longer necessary to provide separate primitives to synchronize the top half and the bottom half. In particular, the *spl* primitives are no longer needed. For compatibility reasons, the calls have been retained, but they translate to no-ops.
- The action of scheduling another process takes significantly longer than interrupt overhead, which also remains.
- The UNIX approach to scheduling does not allow preemption if the process is running in kernel mode.

SMPng solves the latency and scheduling issues with a technique known as *lazy scheduling*: on receiving an interrupt, the interrupt stubs note the PID 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 in the following circumstances:

- If the thread has to block.
- If the interrupt nesting level gets too deep.

This method should offer negligible overhead for the majority of interrupts.

From a scheduling viewpoint, the threads differ from normal processes in the following ways:

- They never enter user mode, so they do not have user text and data segments.
- They all share the address space of process 0, the swapper.
- They run at a higher priority than all user processes.
- Their priority is not adjusted based on load: it remains fixed.
- An additional process state `SWAIT` has been introduced for interrupt processes which are currently idle: the normal "idle" state is `SSLEEP`, which implies that the process is sleeping.

The the initial BSD/OS implementation of interrupt threads was a particularly error-prone process, and that the debugging tools were inadequate. Due to the nature of the FreeBSD project, we considered it imperative to have the system relatively functional at all times during the transition, so we decided to implement interrupt threads in two stages. The initial implementation was very similar to that of normal processes. This offered the benefits of relatively easy debugging and of stability, and the disadvantage of a significant drop in performance: each interrupt could potentially cause two context switches, and the interrupt would not be handled while another process, even a user process, was in the kernel.

Experience with the initial implementation met expectations: we have seen no stability problems with the implementation, and the performance, though significantly worse, was not as bad as we had expected.

At the time of writing, we have improved the implementation somewhat by allowing limited kernel preemption, allowing interrupt threads to be scheduled immediately rather than having to wait for the current process to leave kernel mode. The potential exists for complete kernel preemption, where any higher priority process can preempt a lower priority process running in the kernel, but we are not sure that the benefits will outweigh the potential bug sources.

A *lazy scheduling* implementation has been tested, but it is not currently in the `-CURRENT` kernel. Due to the current kernel lock implementation, it would not show any significant performance increase, and problems can be expected as additional kernel components are migrated from under `Giant`. This issue will probably be reimplemented in terms of the FreeBSD KSE (*Kernel Schedulable Entities*) project.

Not all interrupts have been changed to threaded interrupts. In particular, the old *fast interrupts* remain relatively unchanged, with the restriction that they may not use any blocking mutexes. Fast interrupts have typically been used for the serial drivers, and are specific to FreeBSD: BSD/OS has no corresponding functionality.

Locking constructs

The initial BSD/OS implementation defined two basic types of lock, called *mutex*:

- The default locking construct is the *spin/sleep mutex*. This is similar in concept to a semaphore with a count of 1, but the implementation allows spinning for a certain period of time if this appears to be of benefit (in other words, if it is likely that the mutex will become free in less time than it would take to schedule another process), though this feature is not currently in use. It also allows the user to specify that the mutex should *not* spin. If the process cannot obtain the mutex, it is placed on a sleep queue and woken when the resource becomes available.
- An alternate construct is a *spin mutex*. This corresponds to the spin lock which was already present in the system. Spin mutexes are used only in exceptional cases.

The implementation of these locks was derived almost directly from BSD/OS, but has since been modified significantly.

In addition to these locks, the FreeBSD project has included two further locking constructs:

Condition variables are built on top of mutexes. They consist of a mutex and a wait queue. The following operations are supported:

- Acquire a condition variable with `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()` or `cv_timedwait_sig()`.
- Before acquiring the condition variable, the associated mutex must be held. The mutex will be released before sleeping and reacquired on wakeup.
- Unblock one waiter with `cv_signal()`.
- Unblock all waiters with `cv_broadcast()`.
- Wait for queue empty with `cv_waitq_empty`.
- Same functionality available from the `msleep` function.

Shared/exclusive locks, or *sx locks*, are effectively read-write locks. The difference in terminology came from an intention to add additional functionality to these locks. This functionality has not been implemented, so sx locks are still the same thing as read-write locks: they allow access by multiple readers or a single writer.

The implementation of sx locks is relatively expensive:

```
struct sx {
	struct lock_object sx_object;
	struct mtx	sx_lock;
	int		sx_cnt;
	struct cv	sx_shrd_cv;
	int		sx_shrd_wcnt;
	struct cv	sx_excl_cv;
	int		sx_excl_wcnt;
	struct proc	*sx_xholder;
};
```

They should be only used where the vast majority of accesses is shared.

- Create an sx lock with `sx_init()`.
- Attain a read (shared) lock with `sx_slock()` and release it with `sx_sunlock()`.
- Attain a write (exclusive) lock with `sx_xlock()` and release it with `sx_xunlock()`.
- Destroy an sx lock with `sx_destroy`.

Removing the Big Kernel Lock

These modifications made it possible to remove the Big Kernel Lock. The initial implementation replaced it with two mutexes:

- `Giant` is used in a similar manner to the BKL, but it is a blocking mutex. Currently it protects all entry to the kernel, including interrupt handlers. In order to be able to block, it must allow scheduling to continue.
- `sched_lock` is a spin lock which protects the scheduler queues.

This combination of locks supplied the bare minimum of locks necessary to build the new framework. In itself, it does not improve the performance of the system, since processes still block on Giant.

Idle processes

The planned light-weight interrupt threads need a process context in order to work. In the traditional UNIX kernel, there is not always a process context: the pointer `curproc` can be `NULL`. SMPng solves this problem by having an *idle process* which runs when no other process is active.

Recursive locking

Normally, if a lock is locked, it cannot be locked again. On occasions, however, it is possible that a process tries to acquire a lock which it already holds. Without special checks, this would cause a deadlock. Many implementations allow this so-called *recursive locking*. The locking code checks for the owner of the lock. If the owner is the current process, it increments a recursion counter. Releasing the lock decrements the recursion counter and only releases the lock when the count goes to zero.

There is much discussion both in the literature and in the FreeBSD SMP project as to whether recursive locking should be allowed at all. In general, we have the feeling that recursive locks are evidence of untidy programming. Unfortunately, the code base was never designed for this kind of locking, and in particular library functions may attempt to reacquire locks already held. We have come to a compromise: in general, they are discouraged, and recursion must be specifically enabled for each mutex, thus avoiding recursion where it was not intended.

Migrating to fine-grained locking

Implementing the interrupt threads and replacing the Big Kernel Lock with `Giant` and `schedlock` did not result in any performance improvements, but it provided a framework in which the transition to fine-grained locking could be performed. The next step was to choose a locking strategy and migrate individual portions of the kernel from under the protection of `Giant`.

One of the dangers of this approach is that locking conflicts might not be recognized until very late. In particular, the FreeBSD project has different people working on different kernel components, and it does not have a strong centralized architectural committee to determine locking strategy. As a result, we developed the following guidelines for locking:

- Use sleep mutexes. Spin mutexes should only be used in very special cases and only with the approval of the SMP project team. The only current exception to this rule is the scheduler lock, which by nature must be a spin lock.
- Do not `tsleep()` while holding a mutex other than `Giant`. The implementation of `tsleep()` and `cv_wait()` automatically releases `Giant` and gains it again on wakeup, but no other mutexes will be released.
- Do not `msleep()` or `cv_wait()` while holding a mutex other than `Giant` or the mutex passed as a parameter to `msleep()`. `msleep()` is a new function which combines the functionality with atomic release and regain of a specified mutex.
- Do not call a function that can grab Giant and then sleep unless no mutexes (other than possibly Giant) are held. This is a consequence of the previous rules.
- If calling `msleep()` or `cv_wait()` while holding `Giant` and another mutex, `Giant` must be acquired first and released last. This avoids lock order reversals.
- Except for the `Giant` mutex used during the transition phase, mutexes protect data, not code.
- Do not `msleep()` or `cv_wait()` with a recursed mutex. `Giant` is a special case and is handled automagically behind the scenes, so don't pass Giant to these functions.
- Try to hold mutexes for as little time as possible.
- Try to avoid recursing on mutexes if at all possible. In general, if a mutex is recursively entered, the mutex is being held for too long, and a redesign is in order.

One of the weaknesses of the project structure is that there is no overall strategy for locking. In many cases, the choice of locking construct and granularity is left to the individual developer. In almost every case, locks are leaf node locks: 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 be result in optimum performance.

There are a number of reasons why we persist with this approach:

- FreeBSD is a volunteer project. Developers do what they think is best. They are unlikely to agree to an alternative implementation.
- We do not currently have enough architectural direction, nor enough experience with other SMP systems, to come up with an ideal locking strategy. This derives from the volunteer nature of the project, but note also that large UNIX vendors have found the choice of locking strategy to be a big problem.
- Unlike large companies, there is much less concern about throwaway implementations. If we find that the performance of a system component is suboptimal, we will discard it and start with a different implementation.

Migrating interrupt handlers

This new basic structure is now in place, and implementation of finer grained locking is proceeding. Giant will remain as a legacy locking mechanism for code which has not been converted to the new locking mechanism. For example, the main loop of the function `ithread_loop`, which runs an interrupt handler, contains the following code:

```
if ((ih->ih_flags & IH_MPSAFE) == 0)
        mtx_lock(&Giant);
....
ih->ih_handler(ih->ih_argument);
if ((ih->ih_flags & IH_MPSAFE) == 0)
        mtx_unlock(&Giant);
```

The flag `INTR_MPSAFE` indicates that the interrupt handler has its own synchronization primitives.

A typical strategy planned for migrating device drivers involves the following steps:

- Add a mutex to the driver `softc`.
- Set the `INTR_MPSAFE` flag when registering the interrupt.
- Obtain the mutex in the same kind of situation where previously an `spl` was used. Unlike `spl`s, however, the interrupt handlers must also obtain the mutex before accessing shared data structures.

Probably the most difficult part of the process will involve larger components of the system, such as the file system and the networking stack. We have the example of the BSD/OS code, but it's currently not clear that this is the best path to follow.

Decisions at Yahoo!

After some discussion, we agreed to base the new SMP implementation on BSDi's SMPng implementation. The first question was how to proceed with the imports. It turned out that not many people wanted to participate. From the minutes of the meeting, we find:

- Matt Dillon will put in locking primitives and schedlock. This includes resurrecting our long-dead idle process to scan the run queue for interrupt threads. He won't have time for NFS.
- Greg Lehey will implement the heavyweight interrupt processes and lightweight interrupt threads.
- Doug Rabson will do Alpha machine-dependent work and keep in touch with i386 work. Doesn't want to commit to doing NFS.
- Paul Saab will add kdebug functionality to our DDB.
- Jonathan Lemon wants to convert network drivers.
- Chuck Paterson will be active in a passive role to support the project. He expects to spend about 50% of his time doing so.
- Jason Evans will be the project manager, the first time we've had anybody doing this job.

Of these, only Matt, Greg and Jason had specific tasks assigned.

Stability

We knew that the changes necessary to make SMPng would destabilize the system for a considerable period of time. Jason announced the project on 19 June 2000 with the following summary:

> Summary: -current will be destabilized for an extended period (on the order of months). A tag (not a branch) will be laid down before the initial checkin, and non-developers should either stick closely to that tag until the kernel stabilizes, or expect large doses of pain. This tag will be laid down as soon as June 26, 00:00 PST, with a minimum 24 hour warning beforehand.

This caused some concern in the developer community. Warner Losh replied:

> A few days or weeks I don't have a prblem with, but a few months is flat not acceptible. It is too long. If the code is that green, then some other mechanism needs to be used to facilitate collaberative working.
>
> I'd rather see a firm deadline proposed (eg, we'll commit the core on June 26, and will be done by Aug 26) so that I know what to expect rather than having the nebulous a few months phrase kicked around.

After considerable discussion, we agreed to delay the initial commit until the code was at least vaguely functional. In the interim we maintained giant patch sets, which were very painful.

Initial progress

Matt Dillon did the initial work, not exactly as planned. He ran into significant difficulties with the difference between BSD/OS and FreeBSD, and due to time constraints he ended up doing his own implementation, which was not what we had agreed on. One of the guidelines which we had established was to use BSDi's code wherever possible in order to introduce as few bugs as possible. Ultimately a lot of much of Matt's work was re-imported as intended. Matt completed his work on 25 June 2000, only a few days after the meeting.

Next, still in single threaded mode, Greg Lehey (myself) changed the interrupt model to a "heavyweight thread" approach. A heavyweight thread is effectively a high-priority process, so interrupts would cause context switches. The final goal is to have light-weight interrupt threads which would only context switch if they needed to wait on something, or if they interrupted another interrupt thread. We chose this sub-optimal approach for reasons of stability: BSDi had tried each on different platforms (heavy-weight on SPARC, light-weight on Intel), and they found that debugging was greatly simplified when using heavy-weight interrupt threads.

I had originally estimated two weeks for converting the interrupt system, based on the fact that we already had functional code which we could import from BSD/OS. In fact, it took two months, during which people couldn't do much other work. It's instructive to look at the reasons for this delay:

- BSD/OS and FreeBSD are both derived from 4.4BSD, the last version of BSD UNIX developed at the Computer Sciences Research Group of the University of California at Berkeley. As such, they share a great deal of code, and it's relatively easy for a kernel programmer familiar with one operating system to read, recognize and understand the code of the other operating system. The same applies to NetBSD and OpenBSD, which are also derived from 4.4BSD.
- 4.4BSD did not address a number of the issues facing modern operating systems. In particular, the manner in which hardware components are identified ("probing"), though still more advanced than other UNIX variants, was relatively primitive, and both FreeBSD and BSD/OS had modified it significantly, and of course differently. Again, this issue exists in a similar manner with NetBSD and OpenBSD, though in this case the difference is less, since developers regularly compare code.
- The performance of the 4.4BSD interrupt system was suboptimal. FreeBSD introduced a new kind of interrupt, a *fast interrupt*, to address this problem. BSD/OS does not have fast interrupts.
- Even the BSD/OS code was incompatible between SPARC and Intel. Separate programmers had written the code for each platform, resulting not only in different implementations, but also different names for the same function.
- We were importing the code for heavy-weight interrupt threads, which was only in the SPARC port.

I spent something like a month comparing four different code bases: BSD/OS 4.0 (their old implementation), BSD/OS SMPng Intel, BSD/OS SMPng SPARC, and FreeBSD 4-CURRENT. In this time, I learnt a lot:

- Importing kernel code from different operating systems, even closely related systems, is significantly more difficult than most people understand. I became quite sympathetic to the views of people who didn't want to merge BSD/OS and FreeBSD.
- I also became quite sympathetic to Matt Dillon's violation of our agreement to import the BSD/OS code. He was working on time constraint, and unlike me, he met his deadline. It's difficult to say whether his approach was correct, but it was certainly very understandable.
- I again appreciated the availability of a multiple monitor X server. I had multiple editor windows up on each of three monitors, and I had to ensure that I only edited a specific code base in specific windows: the code was so similar that even so I frequently confused one code base for another.

Finishing this work was immensely satisfying: finally we had something obvious to distinguish the system from any other UNIX or UNIX-like system. For example, the interrupt processes are visible in a *ps* listing:

```
USER      PID %CPU %MEM   VSZ  RSS  TT  STAT STARTED      TIME COMMAND
root       11 99.0  0.0     0   12  ??  RL    2:11PM 150:09.29  (idle: cpu1)
root       12 99.0  0.0     0   12  ??  RL    2:11PM 150:08.71  (idle: cpu0)
root        1  0.0  0.3   756  384  ??  ILs   2:11PM   0:00.11 /sbin/init --
root       13  0.0  0.0     0   12  ??  WL    2:11PM   0:30.61  (swi8: tty:sio clock)
root       15  0.0  0.0     0   12  ??  WL    2:11PM   0:01.32  (swi1: net)
root        2  0.0  0.0     0   12  ??  DL    2:11PM   0:02.35  (g_event)
root        3  0.0  0.0     0   12  ??  DL    2:11PM   0:01.53  (g_up)
root        4  0.0  0.0     0   12  ??  DL    2:11PM   0:01.63  (g_down)
root       16  0.0  0.0     0   12  ??  DL    2:11PM   0:01.05  (random)
root       17  0.0  0.0     0   12  ??  WL    2:11PM   0:00.16  (swi7: task queue)
root       18  0.0  0.0     0   12  ??  WL    2:11PM   0:00.00  (swi6:+)
root        5  0.0  0.0     0   12  ??  DL    2:11PM   0:00.00  (taskqueue)
root       21  0.0  0.0     0   12  ??  WL    2:11PM   0:00.01  (swi3: cambio)
root       22  0.0  0.0     0   12  ??  WL    2:11PM   0:00.08  (irq14: ata0)
root       24  0.0  0.0     0   12  ??  WL    2:11PM   0:02.38  (irq2: rl0 uhci0)
root       25  0.0  0.0     0   12  ??  DL    2:11PM   0:00.00  (usb0)
root       26  0.0  0.0     0   12  ??  DL    2:11PM   0:00.00  (usbtask)
root       27  0.0  0.0     0   12  ??  WL    2:11PM   0:00.00  (irq5: fwohci0++)
root       28  0.0  0.0     0   12  ??  WL    2:11PM   0:00.00  (irq10: sym0)
root       29  0.0  0.0     0   12  ??  WL    2:11PM   0:00.01  (irq11: sym1)
root       30  0.0  0.0     0   12  ??  WL    2:11PM   0:00.00  (irq1: atkbd0)
root       31  0.0  0.0     0   12  ??  WL    2:11PM   0:00.00  (irq6: fdc0)
root        6  0.0  0.0     0   12  ??  DL    2:11PM   0:00.04  (pagedaemon)
root        7  0.0  0.0     0   12  ??  DL    2:11PM   0:00.00  (vmdaemon)
root       39  0.0  0.0     0   12  ??  DL    2:11PM   0:01.00  (syncer)
```

For the first time, we also had the ability to measure the time used by each interrupt.

The interrupt threads were finally committed to the FreeBSD source tree on 6 September 2000. After this, more parallel work could begin.

Progress after September 2000

By the end of September, the new system was running, though a number of areas were particularly fragile. Jason Evans, the project manager, had a particularly delicate task to handle: unlike a commercial project manager, he could not assign tasks to the people whom he found most suitable, nor require that anything be finished by a specific data. Managing volunteer projects is difficult at the best of times.

Early on, John Baldwin had joined the group, to be followed by Jake Burkholder. Neither of these people had been at the meeting, but they ended up doing more and more of the work. John was a BSDi employee, and as a result of his interest was given time to work on the project; Jake was a student in Toronto, and none of us had ever met him. The only people who had been at the meeting who were still working on the project were Jason Evans, Doug Rabson and myself. Doug was doing some coding for Alpha support, and Jason wasn't coding.

As the year continued, John and Jake became the mainstay of the project. I had other demands on my time, and the only outstanding item I had to do was to write the light-weight threads. For a number of reasons, we decided to postpone this particular item.

A lot of the work done during this time was of cosmetic nature. The locking code which we had imported from BSD/OS was very difficult to read, and a rewrite was definitely in order; the question was whether the time was ripe. At the same time a number of functions were moved from one file to another, and their names changed. This is not necessarily factor unique to open source development, of course: very similar things happened within BSDi between the SPARC and Intel ports.

Round about this time, the project appears to have lost focus. The basic structure was in place for the next step, which involved investigating how to implement locking of the various subsystems of the kernel. At this stage, we had only one kind of locking construct, confusingly called a *mutex*, in fact an adaptive lock. We had discussed other locking constructs at the meeting at Yahoo!, in particular condition variables and reader-writer locks, and had come to the conclusion that we should first complete the implementation of the existing system before worrying about other locking constructs. The problem was that this did not suit some of the participants, and so they implemented the constructs they wanted.

We had already established that terminology was one of the big issues in discussing lock nomenclature. The term *mutex* can be applied to all locking constructs, but we chose to use it to describe an adaptive lock. For reasons left to individuals, we ended up with more unusual names. Thus the implementation of reader/writer locks started off being called *sex_lock*, later changed to *sx_lock*, for *shared/exclusive* lock.

In the new year, Jason Evans found that he did not have time for project management. On 5 March 2001, he wrote:

> On another note, I have accepted a job at Sendmail, and am stepping down as SMP project manager due to anticipated time constraints. The SMP project has been in progress for 8 months, and we have 3 to 4 months until the focus of the project needs to shift from development of functionality to performance and stability improvements. There are plenty of disjoint tasks that can be picked up by developers not currently involved in the SMP project. If you want FreeBSD 5.0 to be a success, please consider what you can do to help make it so. There are several unassigned tasks on the task list, and plenty more in the minds of the SMP developers.

There was no replacement project manager. At this stage, a relatively large number of people were working on small parts of the project. Communication was probably better than in a commercial project, but without a project manager it was difficult to coordinate their activities. What coordination went on was done by consensus on IRC channels or on mailing lists. In a number of cases, a developer committed code without review and broke the system for several days.

IRC is not an ideal communications medium for this kind of project. By virtue of its immediate nature, it requires people to be paying attention all the time if they are to gain anything from it. The project is spread round the Earth, which makes this impractical because of time zone differences. It's possible to read the log files, but the volume makes this impractical. In addition, the anarchistic nature of the medium makes it difficult to discuss things in a calm manner. One Linux core developer joined the channel and presented some good views, but one participant decided that he didn't want to see him on that channel and banned him. Over the course of time, the core developers became those who were at home on IRC.

Testing and Debugging

Part of planning a development project is attention to the entire project, not just the software development. Such projects include a QA team to ensure that the testing covers the entire product and not just individual components. Other team members address debugging and performance.

The SMPng project has not addressed these issues effectively. The BSD/OS port included a number of debugging aids. FreeBSD adopted those which were easy to port and omitted the rest, in particular a large part of the kernel trace routines. The project would benefit from more debugging tools, but they're not the kind of activity which attracts people, so it isn't getting done. This is disappointing in some ways, because there are plenty of capable "junior hackers" who could perform this kind of task, but for some reason, probably lack of coordination or communication, nobody has stepped forward to do it.

There's an interesting paradox here: in a commercial environment it would probably be easier to get the tools written, but in an open source environment it's possible that the quality of the tools would be better, since the person writing them would be identified with the quality of the tools. In a commercial environment, once the tools were usable, the project manager would find something else for the person to do.

Documentation

Traditionally, kernel code documentation has been poor. This is not the case for SMPng: the lead developers have been very conscientious about writing man pages for the individual functions. This is an interesting contrast to the debug tools, and it is probably due to personal preferences. Unfortunately, the documentation addresses only the details: overall project documentation is scarce, and this makes it more difficult for newcomers to join the project.

Performance

Another issue that has not been actively examined is performance. The reason for the rewrite is to improve performance on SMP systems. This poses a number of questions:

- What effects does this have on the single processor systems which still make up the bulk of the installed base? The expectation is that the changes will have minimal impact, and that it may be positive.
- How many processors will the implementation support? This is a rather meaningless metric, since almost any implementation, even the old "Big Kernel Lock" implementation, will gain some advantage from any number of additional processors, and it's very dependent on the application. Nevertheless, it is a question often asked. In view of the massively parallel processor architectures currently under development, however, it's a very important one. Putting it more specifically, will SMPng support 32 processor SMP systems? Again, we can't know the answer, but the feeling is that it will probably not do very well against commercial UNIX implementations such as AIX.
- How much good will the light-weight threads do at the moment? The intention was to start the implementation of light-weight threads as soon as the heavyweight threads were stable, but this got deferred based on the assumption that, since nearly all interrupt threads needed to get the `Giant` lock, this would normally involve scheduling anyway. This is a case which could be tested now, but which would involve a lot of possibly unnecessary work.

The only answer we have at the moment is that the performance is notably worse than the performance of the old implementation, in the order of 20% to 30%.

Stability

FreeBSD is recognized as a very stable operating system. How well does the current SMPng implementation live up to this reputation? For some time the answer was "not very well". For some time, there were a number of problems running on specific hardware, but these now seem to have been fixed, and FreeBSD release 5 is at least as stable as its predecessors.

When should we release it?

After SMPng had been in the -CURRENT branch of the FreeBSD source tree for a year or so, plans were made to release it, ready or not. At a "kernel summit" in June 2001 we decided to release an "early adopter" version at the end of the year. In fact, FreeBSD release 5.0 was finally released—still as an "early adopter" version—in December 2002. The intention was that the people who use it would report bugs and enable us to fix them before it became mainstream. FreeBSD release 4 has not yet reached end-of-life: at the time of writing, we're preparing to release version 4.9, and it's possible that a release 4.10 will follow. In a month or two, we will release FreeBSD 5.2.

Summary

SMPng is a work in progress. It's too early to know how successful it will be. Commercial UNIX systems have spent years refining SMP support, so it's not fair to criticize the current state of the system. There are a number of possible ways that the project could develop: it could carry on and be an absolute success, or it could founder and eventually die from structural incompatibilities. In all likelihood, it will be quite successful.

Acknowledgements

The FreeBSD SMPng project was made possible by BSDi's generous donation of code from the development version 5.0 of BSD/OS. The main contributors to the all-important changeover were:

- *John Baldwin* rewrote the low level interrupt code for i386 SMP, made much code machine independent, worked on the `WITNESS` code, converted `allproc` and `proctree` locks from *lockmgr* locks to *sx* locks, created a mechanism in `cdevsw` structure to protect thread-unsafe drivers, locked struct `proc` and unified various SMP API's such as IPI delivery.
- *Jake Burkholder* ported the BSD/OS locking primitives for i386, implemented `msleep()`, condition variables and kernel preemption.
- *Matt Dillon* converted the Big Kernel spinlock to the blocking `Giant` lock and added the scheduler lock and per-CPU idle processes.
- *Jason Evans* made malloc and friends thread-safe, converted simplelocks to mutexes and implemented *sx* (shared/exclusive) locks.
- *Greg Lehey* implemented the heavy-weight interrupt threads, rewrote the low level interrupt code for i386 UP, removed *spl*s and ported the BSD/OS *ktr* code.
- *Bosko Milekic* made *sf_bufs* thread-safe, cleaned up the mutex API and made the *mbuf* system use condition variables instead of `msleep()`.

- *Doug Rabson* ported the BSD/OS locking primitives. implemented the heavy-weight interrupt threads and rewrote the low level interrupt code for the Alpha architecture.

Further contributors were Tor Egge, Seth Kingsley, Jonathan Lemon, Mark Murray, Chuck Paterson, Bill Paul, Alfred Perlstein, Dag-Erling Smørgrav and Peter Wemm.

Bibliography

Per Brinch Hansen, *Operating System Principles*. Prentice-Hall, 1973.

Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley 1996.

Curt Schimmel, *UNIX Systems for Modern Architectures*, Addison-Wesley 1994.

Uresh Vahalia, *UNIX Internals*. Prentice-Hall, 1996.

Further reference

See the FreeBSD SMP home page at *http://www.FreeBSD.org/smg/*.

References

http://www.microsoft.com/misc/backstage/solutions.htm: "Microsoft.com backstage: Solutions / Best Practices". This URL used to contain the recommendations for the FTP server referred to in the section "The problem". It has since been updated.

http://www.mindcraft.com/whitepapers/nts4rhlinux.html: "Web and File Server Comparison: Microsoft Windows NT Server 4.0 and Red Hat Linux 5.2 Upgraded to the Linux 2.2.2 Kernel" was the original Mindcraft benchmark.

http://www.mindcraft.com/whitepapers/openbench1.html: "Open Benchmark: Windows NT Server 4.0 and Linux" is the second benchmark Mindcraft performed on these platforms. It addressed some criticism of the original benchmark by the Linux community.

[USENIX 2001] *http://www.lemis.com/grog/SMPng/USENIX/index.html* The paper "The FreeBSD SMPng implementation" presented at the USENIX summer conference, Boston, 29 June 2001.