Improving the FreeBSD SMP implementation

Greg Lehey
IBM LTC Ozlabs
grog@FreeBSD.org
grog@au1.ibm.com

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 work done to remove this bottleneck, replacing it with fine-grained locking. It derives from work done on BSD/OS and has many similarities with the approach taken in SunOS 5. Synchronization is performed primarily by a locking construct intermediate between a spin lock and a binary semaphore, termed *mutexes*. In general, mutexes attempt to block rather than to spin in cases where the likely wait time is long enough to warrant a process switch. The issue of blocking interrupt handlers is addressed by attaching a process context to the interrupt handlers. Despite this process context, an interrupt handler normally runs in the context of the interrupted process and is scheduled only when blocking is required.

Introduction

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.

Use of processor time

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.

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. The file *sys/kern_fork.c* contains the following code:

```
int
fork1(p1, flags, procp)
        struct proc *p1;
        int flags;
        struct proc **procp;
{
        struct proc *p2, *pptr;

...
        /* Allocate new proc. */
        newproc = zalloc(proc_zone);
```

The function `zalloc` takes a `struct proc` entry off a freelist and returns its address:

```
        item = z->zitems;
        z->zitems = ((void **) item)[0];
...
        return item;
```

What happens if the currently executing process is interrupted exactly between the first two lines of the code above, maybe because a higher priority process wants to run? `item` contains the pointer to the process structure, but `z->z_items` still points to it. If the interrupting code also allocates a process structure, it will go through the same code and return a pointer to the same memory area, 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

Each half must protect its data against change by the other half. For example, the buffer header structure contains a flags word with 32 flags, some set and reset by both halves. Setting and resetting bits requires multiple instructions on most architectures, so the potential for data corruption exists. UNIX solves this problem by locking out interrupts during critical sections. Top half code must explicitly lock out interrupts with the `spl` functions.[1] One of the most significant sources of bugs in drivers is inadequate synchronization with the bottom half.

Interrupt code does not need to perform any special synchronization: by definition, processes don’t run when interrupt code is active.

Blocking interrupts has a potential danger that an interrupt will not be serviced in a timely fashion. On PC hardware, this is particularly evident with serial I/O, which frequently generates an interrupt for every character. At 115200 bps, this equates to an interrupt every 85 μs. In the past, this has given rise to the dreaded silo overflows; even on fast modern hardware it can be a problem. It’s also not easy to decide interrupt priorities: in the early days, disk I/O was given a high priority in order to avoid overruns, while serial I/O had a low priority. Nowadays disk controllers can handle transfers by themselves, but overruns are still a problem with serial I/O.

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

1. The naming goes back to the early days of UNIX on the PDP-11. The PDP-11 had a relatively simplistic level-based interrupt structure. When running at a specific level, only higher priority interrupts were allowed. UNIX named functions for setting the interrupt priority level after the PDP-11 `SPL` instruction, so initially the functions had names like `spl4` and `spl7`. Later machines came out with interrupt masks, and BSD changed the names to more descriptive names such as `splbio` (for block I/O) and `splhigh` (block out all interrupts).

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`, though current BSD no longer uses `sleep`: it has been replaced with `tsleep`, which offers additional functionality.

The top half of a driver calls `sleep` or `tsleep` when it wants to wait for an event, and the bottom half calls `wakeup` when the event occurs. In more detail,

- The process issues a system call `read`, which brings it into kernel mode.
- `read` locates the driver for the device and calls it to initiate a transfer.
- `read` next calls `tsleep`, passing it the address of some unique object related to the request. `tsleep` stores the address in the proc structure, marks the process as sleeping and relinquishes the processor. At this point, the process is sleeping.
- At some later point, when the request is complete, the interrupt handler calls `wakeup` with the address which was passed to `tsleep`. `wakeup` runs through a list of sleeping processes and wakes all processes waiting on this particular address.

This method has problems even on single processors: the time to wake processes depends on the number of sleeping processes, which is usually only slightly less than the number of processes in the system. FreeBSD addresses this problem with 128 hashed sleep queues, effectively diminishing the search time by a factor of 128. A large system might have 10,000 processes running at the same time, so this is only a partial solution.

In addition, it is permissible for more than one process to wait on a specific address. In extreme cases dozens of processes wait on a specific address, but only one will be able to run when the resource becomes available; the rest call `tsleep` again. The term *thundering horde* has been devised to describe this situation. FreeBSD has partially solved this issue with the `wakeup_one` function, which only wakes the first process it finds. This still involves a linear search through a possibly large number of process structures, and it has the potential to deadlock if two unrelated events map to the same address.

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. Code 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.

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:

```
MPgetlock_edx:
1:
        movl    (%edx), %eax
        movl    %eax, %ecx
        andl    $CPU_FIELD,%ecx
        cmpl    _cpu_lockid, %ecx
        jne     2f
        incl    %eax
        movl    %eax, (%edx)
        ret
2:
        movl    $FREE_LOCK, %eax
        movl    _cpu_lockid, %ecx
        incl    %ecx
        lock
        cmpxchg %ecx, (%edx)
        jne     1b
        GRAB_HWI
        ret
```

In an extreme case, this waiting could degrade SMP performance to below that of a single processor machine.

How to solve the dilemma

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. At least the following synchronization primitives have been used in the past:

- *Counting semaphores* were originally designed to share a certain number of resources amongst potentially more consumers. To get access, a consumer decrements the semaphore counter, and when it is finished it increments it again. If the semaphore counter goes negative, the process is placed on a *sleep queue*. If it goes from -1 to 0, the first process on the sleep queue is activated. This approach is a possible alternative to `tsleep` and `wakeup` synchronization. In particular, it avoids a lengthy sequential search of sleeping processes.

- SunOS 5 uses *turnstiles* to address the sequential search problem in `tsleep` and `wakeup` synchronization. A turnstile is a separate queue associated with a specific wait address, so the need for a sequential search disappears.

- *Spin locks* have already been mentioned. FreeBSD used to spin indefinitely on the BKL, which doesn't make any sense, but they are useful in cases where the wait is short; a longer wait will result in a process being suspended and subsequently rescheduled. If the average wait for a resource is less than this time, then it makes sense to spin instead.

- *Blocking locks* are the alternative to spin locks when the wait is likely to be longer than it would take to reschedule. A typical implementation is similar to a counting semaphore with a count of 1.

- *Condition variables* are a kind of blocking lock where the lock is based on a condition, for example the absence of entries in a queue.

- *Read/write locks* address a different issue: frequently multiple processes may read specific data in parallel, but only one may write it.

There is some confusion in terminology with these locking primitives. In particular, the term *mutex* has been applied to nearly all of them at different times. We'll look at how FreeBSD uses the term in the next section.

One big problem with all locking primitives with the exception of spin locks is that they can block. This requires a process context: a UNIX interrupt handler can't block. This is one of the reasons that the old BKL was a spinlock, even though it could potentially use up most of processor time spinning.

The new FreeBSD implementation

The new implementation of SMP on FreeBSD bases heavily on the implementation in BSD/OS 5.0, which has not yet been released. Even the name *SMPng* ("new generation") was taken from BSD/OS. Due to the open source nature of FreeBSD, SMPng is available on FreeBSD before on BSD/OS.

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.

We expect this method to 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.

Experience with the BSD/OS implementation showed that the initial 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.

The final *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`.

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 currently sx locks are 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 `sched-lock` 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 `spls`, 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.

Kernel trace facility

The *ktr* package provides a method of tracing kernel events for debugging purposes. It is not intended for use during normal operation, and should not be confused with the kernel call trace facility *ktrace*.

For example, the function `sched_ithd`, which schedules the interrupt threads, contains the following code:

```
CTR3(KTR_INTR,
     "sched_ithd pid %d(%s) need=%d",
     ir->it_proc->p_pid,
     ir->it_proc->p_comm,
     ir->it_need);
...
if (ir->it_proc->p_stat == SWAIT) {
        CTR1(KTR_INTR,
             "sched_ithd: setrunqueue %d",
            ir->it_proc->p_pid);
```

The function `ithd_loop`, which runs the interrupt in process context, contains the following code at the beginning and end of the main loop:

```
for (;;) {
        CTR3(KTR_INTR,
             "ithd_loop pid %d(%s) need=%d",
            me->it_proc->p_pid,
             me->it_proc->p_comm,
             me->it_need);
...
        CTR1(KTR_INTR,
            "ithd_loop pid %d: done",
             me->it_proc->p_pid);
        mi_switch();
        CTR1(KTR_INTR,
             "ithd_loop pid %d: resumed",
             me->it_proc->p_pid);
```

The calls `CTR1` and `CTR3` are two macros which only compile any kind of code when the kernel is built with the `KTR` kernel option. If the kernel contains this option and the bit `KTR_INTR` is set in the variable `ktr_mask`, then these events will be masked to a circular buffer in the kernel. The *ddb* debugger has a command *show ktr* which dumps the buffer one page at a time, and *gdb* macros are also available. This gives a relatively useful means of tracing the interaction between processes:

```
2791 968643993:219224100
      cpu1 ../../i386/isa/ithread.c:214
      ithd_loop pid 21 ih=0xc235f200:
      0xc0324d98(0) flg=100
2790 968643993:219214043
      cpu1 ../../i386/isa/ithread.c:197
      ithd_loop pid 21(irq0: clk) need=1
2789 968643993:219205383
      cpu1 ../../i386/isa/ithread.c:243
      ithd_loop pid 21: resumed
2788 968643993:219190856
      cpu1 ../../i386/isa/ithread.c:158
      sched_ithd: setrunqueue 21
2787 968643993:219179402
      cpu1 ../../i386/isa/ithread.c:120
      sched_ithd pid 21(irq0: clk) need=0
```

The lines here are too wide for the paper, so they are shown wrapped as several lines. This example traces the arrival and processing of a clock interrupt on the i386 platform, in reverse chronological order. The number at the beginning of the line is the trace entry number.

- Entry 2787 shows the arrival of an interrupt at the beginning of `sched_ithd`. The second value on the trace line is the time since the epoch, followed by the CPU number and the file name and line number. The remaining values are supplied by the program to the `CTR3` function.
- Entry 2788 shows the second trace call in `sched_ithd`, where the interrupt handler is placed on the run queue.

- Entry 2789 shows the entry into the main loop of `ithd_loop`.
- Entries 2790 and 2791 show the exit from the main loop of `ithd_loop`.

Witness facility

The *witness* code was designed specifically to debug mutex code. It keeps track of the locks acquired and released by each thread. It also keeps track of the order in which locks are acquired with respect to each other. Each time a lock is acquired, witness uses these two lists to verify that a lock is not being acquired in the wrong order. If a lock order violation is detected, then a message is output to the kernel console detailing the locks involved and the locations in question. Witness can also be configured to drop into the kernel debugger when an order violation occurs.

The witness code also checks various other conditions such as verifying that one does not recurse on a non-recursive lock. For sleep locks, witness verifies that a new process would not be switched to when a lock is released or a lock is blocked on during an acquire while any spin locks are held. If any of these checks fail, the kernel will panic.

Project status

The project started in June 2000. The major milestones in the development are:

- June 2000: Ported the BSD/OS mutex code and replaced the Big Kernel Lock with `Giant` and `sched_lock`.
- September 2000: Replaced interrupt handlers with heavyweight interrupt processors. Initial commit to the FreeBSD source tree.
- November 2000: Made `softclock` MP-safe and migrate from under `Giant`.
- January 2001: Implemented condition variables.
- March 2001: Implemented read/write locks (called "shared/exclusive" or *sx* locks).
- March 2001: Complete locking of enough of the `proc` structure to allow signal handlers to be moved from under `Giant`.

The main issue in the immediate future is to migrate more and more code out from under `Giant`. In more detail, we have identified the following major tasks, some of which are in an advanced state of implementation:

- Split NFS into client and server.
- Add locking to NFS.
- Make the IP stack thread-safe.
- Create mechanism in `cdevsw` structure to protect thread-unsafe drivers.
- Complete locking struct proc.
- Cleanup the various mp_machdep.c's, unify various SMP API's such as IPI delivery, etc.
- Make `printf()` safe to call in almost any situation to avoid deadlocks.
- Make mbuf system use condition variables instead of `msleep()` and `wakeup()`.
- Remove the MP safe syscall flag from the system call table and add explicit `mtx_lock` of `Giant` to all system calls which need it.
- Use per-CPU buffers for *ktr* to reduce synchronization.
- Remove the priority argument from `msleep()` and `cv_wait()`.
- Implement lazy interrupt thread switching (context stealing).
- Lock structs `filedesc`, `pgrp`, `sigio`, `session` and `ifnet`.
- Make the virtual memory subsystem thread-safe.
- Convert `select()` to use condition variables.
- Reimplement *kqueue* using condition variables.
- Conditionalize atomic operations used for debugging statistics.
- Lock the virtual file system code.

Performance

The implementation has not progressed far enough to make any firm statements about performance, but we are expecting reasonable scalability to beyond 32 processor systems.

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 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/*.