Improving the FreeBSD SMP implementation

Greg Lehey
Nan Yang Computer Services Ltd.
PO Box 460
Echunga SA 5153
grog@lemis.com

11 September 2000

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. It derives from work done on BSD/OS and has many similarities with the approach taken in SunOS 5. Synchronization is performed by *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.

This paper is a snapshot of a work in progress. An up-to-date version is available at *http://www.lemis.com/SMPng/*.

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. 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:

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 without some kind of synchronization the data would be corrupted. UNIX performs this synchronization 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 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. UNIX performs this synchronization with the functions `sleep` and `wakeup`.[2] 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,

---

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).

2. FreeBSD no longer uses `sleep`, having replaced it with `tsleep`, which offers additional functionality.

- 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.

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            /* Get current contents of lock */
        movl    %eax, %ecx
        andl    $CPU_FIELD,%ecx
        cmpl    _cpu_lockid, %ecx       /* Do we already own the lock? */
        jne     2f
        incl    %eax                    /* yes, just bump the count */
        movl    %eax, (%edx)            /* serialization not required */
        ret
2:
        movl    $FREE_LOCK, %eax        /* lock must be free */
        movl    _cpu_lockid, %ecx
        incl    %ecx
        lock
        cmpxchg %ecx, (%edx)            /* attempt to replace %eax<->%ecx */
        jne     1b
        GRAB_HWI                        /* 1st acquire, grab hw INTs */
        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: an interrupt handler can't block. This is one of the reasons that the old BGL 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 is that 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 use of blocking locks requires a process context, so interrupts are now handled as threads. The initial implementation is very similar to normal processes, with the following differences:

- Interrupt processes run in their own scheduling class, which is scheduled ahead of the other three classes which FreeBSD supplies.
- 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 current implementation imposes a scheduling overhead which decreases performance significantly, but it is relatively stable. In the month up to the time when this paper was submitted, we have seen no stability problems with the implementation. At a later date, but before release, we will reimplement interrupt threads in a manner similar to the BSD/OS implementation. This *lightweight threads* implementation involves *lazy scheduling* of the interrupt thread: since normally interrupts interrupt processes and not other interrupts, and since they thus normally run at a higher priority, they can take control of the processor directly. They only need to be scheduled if they have to block. The situation becomes significantly more complicated if interrupts occur while an interrupt

handler is running, in particular regarding relative interrupt priorities. As a result we have decided to do the implementation in two steps, particularly after reports of experience with BSD/OS, which implemented light-weight threads directly.

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.

Mutexes

The mutex implementation defines two basic types of mutex:

- The default mutex is the *spin/sleep mutex*. If the process cannot obtain the mutex, it is placed on a sleep queue and woken when the resource becomes available. This is similar in concept to semaphores, 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). It also allows the user to specify that the mutex should *not* spin.
- Alternatively, a mutex may be defined as a *spin mutex*. In this case, it will never sleep. Effectively, this is the spin lock which was already present in the system.

The mutex implementation was derived almost directly from BSD/OS.

Removing the Big Kernel Lock

These modifications made it possible to remove the Big Kernel Lock. The current implementation has 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 supplies 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

Other features

In addition to the basic changes above, a number of debugging aids were ported from BSD/OS:

- The *ktr* package provides a method of tracing kernel events. 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) { /* not on run queue */
                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. Currently *gdb* macros are available to decode them, giving 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 two 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`.
- The *witness* code was designed specifically to debug mutex code. At present it is not greatly needed, since there is little scope for deadlocks, but based on the BSD/OS experience we expect it will be of great use in the future.

The future

With this new basic structure in place, implementation of finer grained locking can proceed. 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 `ithd_loop`, which runs an interrupt handler, contains the following code:

```
if ((ih->flags & INTR_MPSAFE) == 0)
        mtx_enter(&Giant, MTX_DEF);
ih->handler(ih->argument);
if ((ih->flags & INTR_MPSAFE) == 0)
        mtx_exit(&Giant, MTX_DEF);
```

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.

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.

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

Greg Lehey: short biography

Greg Lehey is an independent computer consultant specializing in UNIX. Born in Melbourne, Australia, he was educated in Malaysia, Germany and England. He returned to Australia in 1997 after spending most of his professional career in Germany, where he worked for computer manufacturers such as Univac, Tandem, and Siemens-Nixdorf, the German space research agency, nameless software houses and a large user and finally for himself as a consultant. In the course of more than 25 years in the industry he has performed most jobs, ranging from kernel development to product marketing, from systems programming to operating, from processing satellite data to programming petrol pumps, from the production of CD-ROMs of ported free software to DSP instruction set design. He is also a member of the executive committee of the AUUG and the author of ‘‘Porting UNIX Software’’ (O’Reilly and Associates, 1995), ‘‘Installing and Running FreeBSD’’ (Walnut Creek, 1996), and ‘‘The Complete FreeBSD’’ (Walnut Creek, 1997—1999). About the only thing he hasn’t done is writing commercial applications software. Browse his home page at *http://www.lemis.com/~grog/*.

When he can drag himself away from his collection of old UNIX hardware, he is involved in performing baroque and classical woodwind music on his collection of original instruments, exploring the Australian countryside with his family on their Arabian horses, or exploring new cookery techniques or ancient and obscure European languages.