SPINLOCKS, Part II

by Mike Rieker

Simultaneous spinlocks

Now after you get into this stuff, you may find that you have to have more than one spinlock at a time. This can lead to problems. Consider this:

Routine A:

```
  lock spinlock X
  maybe do some work
  lock spinlock Y
  do some more work
  unlock spinlock Y
  unlock spinlock X
```

And Routine B:

```
  lock spinlock Y
  maybe do some work
  lock spinlock X
  do some more work
  unlock spinlock X
  unlock spinlock Y
```

So CPU #1 comes along and starts routine A, while CPU #2 starts routine B. Well, they'll never finish, because CPU #1 will be waiting for CPU #2 to release spinlock Y and CPU #2 will be waiting for CPU #1 to release spinlock X.

So we have another simple rule: When locking more than one spinlock, they must always be locked in the same order.

So for our example, both routines need to be changed so that they either both lock X then Y or they both lock Y then X. You may be thinking, 'I might not be able to do that in ALL cases!' Easy to fix, replace the two spinlocks with one, say Z.

Now I am terrible at checking to make sure I do everything in order. Computers are good at that sort of thing. So rather than just using an 'int' for my spinlocks, I use a struct as follows:

```
  typedef struct { char spinlock_flag;            // the usual 0=unlocked, 1=locked
                   unsigned char spinlock_level;  // 'locking' order
                 } Spinlock;
```

Then I have a spinlock_wait routine that checks to make sure my new spinlock's level is .gt. the last spinlock's level that was locked by this cpu. If I try to do it out of order, the routine panics/BSOD's on the spot.

Interrupts

Another question you may wonder, must I always inhibit hardware interrupt delivery when I have a spinlock?

No. Only if the data being worked with can also be accessed by an interrupt routine. And then, you only have to disable those interrupts that have access to it.

For example, I don't allow my interrupt routines to do malloc's. So the malloc routine can run with interrupts enabled, even when it has the spinlock. Same with pagetables, disk cache pages, and other stuff like that.

You must, however, block any pre-emptive scheduling while you have a any spinlock. Or at least, you will make things *very* complicated if you don't.

So I ended up with several 'classes' of spinlocks:

- low priority - these run with all hardware interrupt delivery enabled
- interrupt level - there is one per IRQ level. IRQ 0 is the highest, IRQ 7 is the lowest when one of these is set on a cpu, it inhibits that IRQ and any lower priority interrupts
- high priority - these run with all hardware interrupt delivery inhibited

So if a cpu has one of the low priority spinlocks, it is running with hardware interrupt delivery enabled (but the pre-emptive scheduler is disabled).

There is one spinlock per interrupt level. For example, the keyboard driver uses IRQ 1. So whenever the keyboard interrupt routine is active, I have the cpu take the IRQ 1 spinlock. Whenever the main keyboard driver routine is accessing data that the interrupt routine would access, I take the IRQ 1 spinlock. So it in essence acts as a 'inhibit interrupt delivery' mechanism that works across cpu's in an SMP system.

The high-priority interrupts block all interrupt delivery on the cpu. These are used for routines (like wake_a_thread) that need to be callable from any interrupt level.

Performance

This is a big thing about spinlocks. It's easy to make your performance go to pot with poor use of spinlocks.

Consider that you can *theoretically* use just *one* spinlock, period. Whenever you enter kernel mode, inhibit all interrupt delivery and set your one-and-only spinlock. When you leave kernel mode, release the spinlock then enable interrupt delivery. Great!

Well, you can imagine then, that what you end up with is having just one cpu doing anything useful at a time in kernel mode, while the other(s) just grind away spinning. Really bad!

So what you do is try to come up with as many spinlocks as you can. Above I mentioned that you must combine X and Y if you have a case where you must set X then Y and another case where you must set Y then X. So now you have two limits to work with:

- Too few spinlocks and you have poor performance
- Too many spinlocks and you violate the ordering

Here's where the creativity comes in, working with those two limits to maximize performance. Strive to eliminate as much spinning as possible.

How does one measure spinlock performance? Make an array of counters like this:

```
  static int spin_counts[256];
```

Then increment spin_counts[spinlock_level] each time the 'test_and_set' fails. After a while of running stuff, find which spinlock_level has big counts (compared to the others). Try to break that smplock down into smaller ones if you can figure out how to.

Summary

They may seem sort of nebulous to begin with. But after you work with them, application becomes very scientific. I hope this little tutorial has helped clear up some issues and gives you confidence.

I have some more practical stuff at www.o3one.org/smplocks.html if you're still interested.

Mike's home page can be found at http://www.o3one.org/