Locking lessons

Lesson 1: Spin locks

The most basic primitive for locking is spinlock:

```
static DEFINE_SPINLOCK(xxx_lock);

      unsigned long flags;

      spin_lock_irqsave(&xxx_lock, flags);
      ... critical section here ..
      spin_unlock_irqrestore(&xxx_lock, flags);
```

The above is always safe. It will disable interrupts _locally_, but the spinlock itself will guarantee the global lock, so it will guarantee that there is only one thread-of-control within the region(s) protected by that lock. This works well even under UP also, so the code does _not_ need to worry about UP vs SMP issues: the spinlocks work correctly under both.

> NOTE! Implications of spin_locks for memory are further described in:
>
> Documentation/memory-barriers.txt
>
> 5. ACQUIRE operations.
> 6. RELEASE operations.

The above is usually pretty simple (you usually need and want only one spinlock for most things - using more than one spinlock can make things a lot more complex and even slower and is usually worth it only for sequences that you **know** need to be split up: avoid it at all cost if you aren't sure).

This is really the only really hard part about spinlocks: once you start using spinlocks they tend to expand to areas you might not have noticed before, because you have to make sure the spinlocks correctly protect the shared data structures **everywhere** they are used. The spinlocks are most easily added to places that are completely independent of other code (for example, internal driver data structures that nobody else ever touches).

> NOTE! The spin-lock is safe only when you **also** use the lock itself to do locking across CPU's, which implies that EVERYTHING that touches a shared variable has to agree about the spinlock they want to use.

Lesson 2: reader-writer spinlocks.

If your data accesses have a very natural pattern where you usually tend to mostly read from the shared variables, the reader-writer locks (rw_lock) versions of the spinlocks are sometimes useful. They allow multiple readers to be in the same critical region at once, but if somebody wants to change the variables it has to get an exclusive write lock.

> NOTE! reader-writer locks require more atomic memory operations than simple spinlocks. Unless the reader critical section is long, you are better off just using spinlocks.

The routines look the same as above:

```
rwlock_t xxx_lock = __RW_LOCK_UNLOCKED(xxx_lock);

     unsigned long flags;

     read_lock_irqsave(&xxx_lock, flags);
     .. critical section that only reads the info ...
     read_unlock_irqrestore(&xxx_lock, flags);

     write_lock_irqsave(&xxx_lock, flags);
     .. read and write exclusive access to the info ...
     write_unlock_irqrestore(&xxx_lock, flags);
```

The above kind of lock may be useful for complex data structures like linked lists, especially searching for entries without changing the list itself. The read lock allows many concurrent readers. Anything that **changes** the list will have to get the write lock.

> NOTE! RCU is better for list traversal, but requires careful attention to design detail (see Documentation/RCU/listRCU.rst).

Also, you cannot "upgrade" a read-lock to a write-lock, so if you at _any_ time need to do any changes (even if you don't do it every time), you have to get the write-lock at the very beginning.

NOTE! We are working hard to remove reader-writer spinlocks in most cases, so please don't add a new one without consensus. (Instead, see Documentation/RCU/rcu.rst for complete information.)

Lesson 3: spinlocks revisited.

The single spin-lock primitives above are by no means the only ones. They are the most safe ones, and the ones that work under all circumstances, but partly **because** they are safe they are also fairly slow. They are slower than they'd need to be, because they do have to disable interrupts (which is just a single instruction on a x86, but it's an expensive one - and on other architectures it can be worse).

If you have a case where you have to protect a data structure across several CPU's and you want to use spinlocks you can potentially use cheaper versions of the spinlocks. IFF you know that the spinlocks are never used in interrupt handlers, you can use the non-irq versions:

```
spin_lock(&lock);
...
spin_unlock(&lock);
```

(and the equivalent read-write versions too, of course). The spinlock will guarantee the same kind of exclusive access, and it will be much faster. This is useful if you know that the data in question is only ever manipulated from a "process context", ie no interrupts involved.

The reasons you mustn't use these versions if you have interrupts that play with the spinlock is that you can get deadlocks:

```
spin_lock(&lock);
...
        <- interrupt comes in:
                spin_lock(&lock);
```

where an interrupt tries to lock an already locked variable. This is ok if the other interrupt happens on another CPU, but it is _not_ ok if the interrupt happens on the same CPU that already holds the lock, because the lock will obviously never be released (because the interrupt is waiting for the lock, and the lock-holder is interrupted by the interrupt and will not continue until the interrupt has been processed).

(This is also the reason why the irq-versions of the spinlocks only need to disable the _local_ interrupts - it's ok to use spinlocks in interrupts on other CPU's, because an interrupt on another CPU doesn't interrupt the CPU that holds the lock, so the lock-holder can continue and eventually releases the lock).

Linus

Reference information:

For dynamic initialization, use spin_lock_init() or rwlock_init() as appropriate:

```
spinlock_t xxx_lock;
rwlock_t xxx_rw_lock;

static int __init xxx_init(void)
{
     spin_lock_init(&xxx_lock);
     rwlock_init(&xxx_rw_lock);
     ...
}

module_init(xxx_init);
```

For static initialization, use DEFINE_SPINLOCK() / DEFINE_RWLOCK() or __SPIN_LOCK_UNLOCKED() / __RW_LOCK_UNLOCKED() as appropriate.