Kernel Korner

Kernel Locking Techniques

Robert explains the various locking primitives in the Linux kernel, why you need them and how kernel developers can use them to write safe code.

by Robert Love

Proper locking can be tough--real tough. Improper locking can result in random crashes and other oddities. Poorly designed locking can result in code that is hard to read, performs poorly and makes your fellow kernel developers cringe. In this article, I explain why kernel code requires locking, provide general rules for proper kernel locking semantics and then outline the various locking primitives in the Linux kernel.

Why Do We Need Locking in the Kernel?

The fundamental issue surrounding locking is the need to provide synchronization in certain code paths in the kernel. These code paths, called critical sections, require some combination of concurrency or re-entrancy protection and proper ordering with respect to other events. The typical result without proper locking is called a race condition. Realize how even a simple i++ is dangerous if i is shared! Consider the case where one processor reads i, then another, then they both increment it, then they both write i back to memory. If i were originally 2, it should now be 4, but in fact it would be 3!

This is not to say that the only locking issues arise from SMP (symmetric multiprocessing). Interrupt handlers create locking issues, as does the new preemptible kernel, and any code can block (go to sleep). Of these, only SMP is considered true concurrency, i.e., only with SMP can two things actually occur at the exact same time. The other situations--interrupt handlers, preempt-kernel and blocking methods--provide pseudo concurrency as code is not actually executed concurrently, but separate code can mangle one another's data.

These critical regions require locking. The Linux kernel provides a family of locking primitives that developers can use to write safe and efficient code.

SMP Locks in a Uniprocessor Kernel

Whether or not you have an SMP machine, people who use your code may. Further, code that does not handle locking issues properly is typically not accepted into the Linux kernel. Finally, with a preemptible kernel even UP (uniprocessor) systems require proper locking. Thus, do not forget: you must implement locking.

Thankfully, Linus made the excellent design decision of keeping SMP and UP kernels distinct. This allows certain locks not to exist at all in a UP kernel. Different combinations of CONFIG_SMP and CONFIG_PREEMPT compile in varying lock support. It does not matter, however, to the developer: lock everything appropriately and all situations will be covered.

Atomic Operators

We cover atomic operators initially for two reasons. First, they are the simplest of the approaches to kernel
synchronization and thus the easiest to understand and use. Second, the complex locking primitives are built off them. In this sense, they are the building blocks of the kernel's locks. Atomic operators are operations, like add and subtract, which perform in one uninterruptible operation. Consider the previous example of i++. If we could read i, increment it and write it back to memory in one uninterruptible operation, the race condition discussed above would not be an issue. Atomic operators provide these uninterruptible operations. Two types exist: methods that operate on integers and methods that operate on bits. The integer operations work like this:

```c
atomic_t v;
atomic_set(&v, 5);      /* v = 5 (atomically) */
atomic_add(3, &v);      /* v = v + 3 (atomically) */
atomic_dec(&v);         /* v = v - 1 (atomically) */
printf("This will print 7: %d\n", atomic_read(&v));
```

They are simple. There are, however, little caveats to keep in mind when using atomics. First, you obviously cannot pass an atomic_t to anything but one of the atomic operators. Likewise, you cannot pass anything to an atomic operator except an atomic_t. Finally, because of the limitations of some architectures, do not expect atomic_t to have more than 24 usable bits. See the "Function Reference" Sidebar for a list of all atomic integer operations.

**Function Reference**

The next group of atomic methods is those that operate on individual bits. They are simpler than the integer methods because they work on the standard C data types. For example, consider void set_bit(int nr, void *addr). This function will atomically set to 1 the "nr-th" bit of the data pointed to by addr. The atomic bit operators are also listed in the "Function Reference" Sidebar.

**Spinlocks**

For anything more complicated than trivial examples like those above, a more complete locking solution is needed. The most common locking primitive in the kernel is the spinlock, defined in include/asm/spinlock.h and include/linux/spinlock.h. The spinlock is a very simple single-holder lock. If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (spinning) until it can acquire the lock. This simplicity creates a small and fast lock. The basic use of the spinlock is:

```c
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);
/* critical section ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

The use of spin_lock_irqsave() will disable interrupts locally and provide the spinlock on SMP. This covers both interrupt and SMP concurrency issues. With a call to spin_unlock_irqrestore(), interrupts are restored to the state when the lock was acquired. With a UP kernel, the above code compiles to the same as:

```c
unsigned long flags;
save_flags(flags);
cli();
/* critical section ... */
restore_flags(flags);
```

which will provide the needed interrupt concurrency protection without unneeded SMP protection. Another variant of the spinlock is spin_lock_irq(). This variant disables and re-enables interrupts unconditionally, in
the same manner as cli() and sti(). For example:

```c
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
spin_lock_irq(&mr_lock);
/* critical section ... */
spin_unlock_irq(&mr_lock);
```

This code is only safe when you know that interrupts were not already disabled before the acquisition of the lock. As the kernel grows in size and kernel code paths become increasingly hard to predict, it is suggested you not use this version unless you really know what you are doing.

All of the above spinlocks assume the data you are protecting is accessed in both interrupt handlers and normal kernel code. If you know your data is unique to user-context kernel code (e.g., a system call), you can use the basic spin_lock() and spin_unlock() methods that acquire and release the specified lock without any interaction with interrupts.

A final variation of the spinlock is spin_lock_bh() that implements the standard spinlock as well as disables softirqs. This is needed when you have code outside a softirq that is also used inside a softirq. The corresponding unlock function is naturally spin_unlock_bh().

Note that spinlocks in Linux are not recursive as they may be in other operating systems. Most consider this a sane design decision as recursive spinlocks encourage poor code. This does imply, however, that you must be careful not to re-acquire a spinlock you already hold, or you will deadlock.

Spinlocks should be used to lock data in situations where the lock is not held for a long time—recall that a waiting process will spin, doing nothing, waiting for the lock. (See the ``Rules'' Sidebar for guidelines on what is considered a long time.) Thankfully, spinlocks can be used anywhere. You cannot, however, do anything that will sleep while holding a spinlock. For example, never call any function that touches user memory, kmalloc() with the GFP_KERNEL flag, any semaphore functions or any of the schedule functions while holding a spinlock. You have been warned.

If you need a lock that is safe to hold for longer periods of time, safe to sleep with or capable of allowing concurrency to do more than one process at a time, Linux provides the semaphore.

**Semaphores**

Semaphores in Linux are sleeping locks. Because they cause a task to sleep on contention, instead of spin, they are used in situations where the lock-held time may be long. Conversely, since they have the overhead of putting a task to sleep and subsequently waking it up, they should not be used where the lock-held time is short. Since they sleep, however, they can be used to synchronize user contexts whereas spinlocks cannot. In other words, it is safe to block while holding a semaphore.

In Linux, semaphores are represented by a structure, struct semaphore, which is defined in include/asm/semaphore.h. The structure contains a pointer to a wait queue and a usage count. The wait queue is a list of processes blocking on the semaphore. The usage count is the number of concurrently allowed holders. If it is negative, the semaphore is unavailable and the absolute value of the usage count is the number of processes blocked on the wait queue. The usage count is initialized at runtime via sema_init(), typically to 1 (in which case the semaphore is called a mutex).

Semaphores are manipulated via two methods: down (historically P) and up (historically V). The former attempts to acquire the semaphore and blocks if it fails. The later releases the semaphore, waking up any tasks blocked along the way.
Semaphore use is simple in Linux. To attempt to acquire a semaphore, call the down_interruptible() function. This function decrements the usage count of the semaphore. If the new value is less than zero, the calling process is added to the wait queue and blocked. If the new value is zero or greater, the process obtains the semaphore and the call returns 0. If a signal is received while blocking, the call returns -EINTR and the semaphore is not acquired.

The up() function, used to release a semaphore, increments the usage count. If the new value is greater than or equal to zero, one or more tasks on the wait queue will be woken up:

```c
struct semaphore mr_sem;
sema_init(&mr_sem, 1); /* usage count is 1 */
if (down_interruptible(&mr_sem))
    /* semaphore not acquired; received a signal ... */
/* critical region (semaphore acquired) ... */
up(&mr_sem);
```

The Linux kernel also provides the down() function, which differs in that it puts the calling task into an uninterruptible sleep. A signal received by a process blocked in uninterruptible sleep is ignored. Typically, developers want to use down_interruptible(). Finally, Linux provides the down_trylock() function, which attempts to acquire the given semaphore. If the call fails, down_trylock() will return nonzero instead of blocking.

**Reader/Writer Locks**

In addition to the standard spinlock and semaphore implementations, the Linux kernel provides reader/writer variants that divide lock usage into two groups: reading and writing. Since it is typically safe for multiple threads to read data concurrently, so long as nothing modifies the data, reader/writer locks allow multiple concurrent readers but only a single writer (with no concurrent readers). If your data access naturally divides into clear reading and writing patterns, especially with a greater amount of reading than writing, the reader/writer locks are often preferred.

The reader/writer spinlock is called an rwlock and is used similarly to the standard spinlock, with the exception of separate reader/writer locking:

```c
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;
read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);
write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_rwlock);
```

Likewise, the reader/writer semaphore is called an rw_semaphore and use is identical to the standard semaphore, plus the explicit reader/writer locking:

```c
struct rw_semaphore mr_rwsem;
init_rwsem(&mr_rwsem);
down_read(&mr_rwsem);
/* critical region (read only) ... */
up_read(&mr_rwsem);
```
Use of reader/writer locks, where appropriate, is an appreciable optimization. Note, however, that unlike other implementations reader locks cannot be automatically upgraded to the writer variant. Therefore, attempting to acquire exclusive access while holding reader access will deadlock. Typically, if you know you will need to write eventually, obtain the writer variant of the lock from the beginning. Otherwise, you will need to release the reader lock and re-acquire the lock as a writer. If the distinction between code that writes and reads is muddled such as this, it may be indicative that reader/writer locks are not the best choice.

**Big-Reader Locks**

Big-reader locks (brlocks), defined in include/linux/brlock.h, are a specialized form of reader/writer locks. Big-reader locks, designed by Red Hat's Ingo Molnar, provide a spinning lock that is very fast to acquire for reading but incredibly slow to acquire for writing. Therefore, they are ideal in situations where there are many readers and few writers.

While the behavior of brlocks is different from that of rwlocks, their usage is identical with the lone exception that brlocks are predefined in brlock_indices (see brlock.h):

```c
br_read_lock(BR_MR_LOCK);
/* critical region (read only) ... */
br_read_unlock(BR_MR_LOCK);
```

Use of brlocks is currently confined to a few special cases. Due to the large penalty for exclusive write access, it should probably stay that way.

**The Big Kernel Lock**

Linux contains a global kernel lock, kernel_flag, that was originally introduced in kernel 2.0 as the only SMP lock. During 2.2 and 2.4, much work went into removing the global lock from the kernel and replacing it with finer-grained localized locks. Today, the global lock's use is minimal. It still exists, however, and developers need to be aware of it.

The global kernel lock is called the big kernel lock or BKL. It is a spinning lock that is recursive; therefore two consecutive requests for it will not deadlock the process (as they would for a spinlock). Further, a process can sleep and even enter the scheduler while holding the BKL. When a process holding the BKL enters the scheduler, the lock is dropped so other processes can obtain it. These attributes of the BKL helped ease the introduction of SMP during the 2.0 kernel series. Today, however, they should provide plenty of reason not to use the lock.

Use of the big kernel lock is simple. Call lock_kernel() to acquire the lock and unlock_kernel() to release it. The routine kernel_locked() will return nonzero if the lock is held, zero if not. For example:

```c
lock_kernel();
/* critical region ... */
unlock_kernel();
```

**Preemption Control**

Starting with the 2.5 development kernel (and 2.4 with an available patch), the Linux kernel is fully preemptible. This feature allows processes to be preempted by higher-priority processes, even if the current
process is running in the kernel. A preemptible kernel creates many of the synchronization issues of SMP. Thankfully, kernel preemption is synchronized by SMP locks, so most issues are solved automatically by writing SMP-safe code. A few new locking issues, however, are introduced. For example, a lock may not protect per-CPU data because it is implicitly locked (it is safe because it is unique to each CPU) but is needed with kernel preemption.

For these situations, preempt_disable() and the corresponding preempt_enable() have been introduced. These methods are nestable such that for each \( n \) preempt_disable() calls, preemption will not be re-enabled until the \( n \)th preempt_enable() call. See the `Function Reference` Sidebar for a complete list of preemption-related controls.

**Conclusion**

Both SMP reliability and scalability in the Linux kernel are improving rapidly. Since SMP was introduced in the 2.0 kernel, each successive kernel revision has improved on the previous by implementing new locking primitives and providing smarter locking semantics by revising locking rules and eliminating global locks in areas of high contention. This trend continues in the 2.5 kernel. The future will certainly hold better performance.

Kernel developers should do their part by writing code that implements smart, sane, proper locking with an eye to both scalability and reliability.

**Rules**

Robert Love (rml@tech9.net) is a Computer Science and Mathematics student at the University of Florida and a kernel engineer at MontaVista Software. Robert is the maintainer of the preemptible kernel and is involved in various other kernel development projects. He loves Jack Handy books and Less than Jake.