Anatomy of Linux synchronization methods

Kernel atomics, spinlocks, and mutexes

M. Tim Jones
Consultant Engineer
Emulex

31 October 2007

In your Linux® education, you may have learned about concurrency, critical sections, and locking, but how do you use these concepts within the kernel? This article reviews the locking mechanisms available within the 2.6 kernel, including atomic operators, spinlocks, reader/writer locks, and kernel semaphores. It also explores where each mechanism is most applicable for building safe and efficient kernel code.

This article explores many of the synchronization or locking mechanisms that are available in the Linux kernel. It presents the application program interfaces (APIs) for many of the available methods from the 2.6.23 kernel. But before you dig into the APIs, you need to understand the problem that's being solved.

More in Tim's *Anatomy of...* series on developerWorks

- All of Tim's *Anatomy of...* articles
- All of Tim's articles on developerWorks

Concurrency and locking

Synchronization methods are necessary when the property of concurrency exists. *Concurrency* exists when two or more processes execute over the same time period and potentially interact with one another (for example, sharing the same set of resources).

Concurrency can occur on uniprocessor (UP) hosts where multiple threads share the same CPU and preemption creates race conditions. *Preemption* is sharing the CPU transparently by temporarily pausing one thread to allow another to execute. A *race condition* occurs when two or more threads manipulate a shared data item and the result depends upon timing of the execution. Concurrency also exists in multiprocessor (MP) machines, where threads executing simultaneously in each processor share the same data. Note that in the MP case there is true parallelism because the threads execute simultaneously. In the UP case, parallelism is created by preemption. The difficulties of concurrency exist in both modes.

The Linux kernel supports concurrency in both modes. The kernel itself is dynamic, and race conditions can be created in a number of ways. The Linux kernel also supports multiprocessing known as symmetric multiprocessing (SMP). You can learn more about SMP in the Resources section later in this article.

To combat the issue of race conditions, the concept of a critical section was created. A *critical section* is a portion of code that is protected against multiple access. This portion of code can manipulate shared data or a shared service (such as a hardware peripheral). Critical sections operate on the principle of mutual exclusion (when a thread is within a critical section, all other threads are excluded from entering).

A problem created within critical sections is the condition of deadlock. Consider two separate critical sections, each protecting a different resource. For each resource, a lock is available, called A and B in this example. Now consider two threads that require access to the resources. Thread X takes lock A, and thread Y takes lock B. While those locks are held, each thread attempts to take the other lock that is currently held by the other thread (thread X wants lock B, and thread Y wants lock A). The threads are now deadlocked because they each hold one resource and want the other. A simple solution is to always take locks in the same order, which allows a thread to complete. Other solutions involve detecting this situation. Table 1 defines important concurrency terms covered here.

Table 1. Important definitions in concurrency

Term	Definition
Race condition	Situation where simultaneous manipulation of a resource by two or more threads causes inconsistent results.
Critical section	Segment of code that coordinates access to a shared resource.
Mutual exclusion	Property of software that ensures exclusive access to a shared resource.
Deadlock	Special condition created by two or more processes and two or more resource locks that keep processes from doing productive work.

Linux synchronization methods

Now that you have a little theory under your belt and an understanding of the problem to be solved, let's look at the various ways that Linux supports concurrency and mutual exclusion. In the early days, mutual exclusion was provided by disabling interrupts, but this form of locking is inefficient (even though you can still find traces of it in the kernel). This method also doesn't scale well and doesn't guarantee mutual exclusion on other processors.

In the following review of locking mechanisms, we first look at the atomic operators, which provide protection for simple variables (counters and bitmasks). Simple spinlocks and reader/writer

spinlocks are then covered as an efficient busy-wait lock for SMP architectures. Finally, we explore kernel mutexes, which are built on top of the atomic API.

Atomic operations

The simplest means of synchronization in the Linux kernel are the atomic operations. *Atomic* means that the critical section is contained within the API function. No locking is necessary because it's inherent in the call. As C can't guarantee atomic operations, Linux relies on the underlying architecture to provide this. Since architectures differ greatly, you'll find varying implementations of the atomic functions. Some are provided almost entirely in assembly, while others resort to C and disabling interrupts using `local_irq_save` and `local_irq_restore`.

Older locking methods

A bad way to implement locking in the kernel is by disabling hard interrupts for the local CPU. These functions are available and are used (sometimes by the atomic operators) but are not recommended. The `local_irq_save` routine disables interrupts, and `local_irq_restore` restores the previously-enabled interrupts. These routines are reentrant, meaning they may be called within context of one another.

The atomic operators are ideal for situations where the data you need to protect is simple, such as a counter. While simple, the atomic API provides a number of operators for a variety of situations. Here's a sample use of the API.

To declare an atomic variable, you simply declare a variable of type `atomic_t`. This structure contains a single `int` element. Next, you ensure that your atomic variable is initialized using the `ATOMIC_INIT` symbolic constant. In the case shown in Listing 1, the atomic counter is set to zero. It's also possible to initialize the atomic variable at runtime using the `atomic_set function`.

Listing 1. Creating and initializing an atomic variable

```
atomic_t my_counter = ATOMIC_INIT(0);

... or ...

atomic_set( &my_counter, 0 );
```

The atomic API supports a rich set of functions covering many use cases. You can read the contents of an atomic variable with `atomic_read` and also add a specific value to a variable with `atomic_add`. The most common operation is to simply increment the variable, which is provided with `atomic_inc`. The subtraction operators are also available, providing the converse of the add and increment operations. Listing 2 demonstrates these functions.

Listing 2. Simple arithmetic atomic functions

```
val = atomic_read( &my_counter );

atomic_add( 1, &my_counter );

atomic_inc( &my_counter );

atomic_sub( 1, &my_counter );

atomic_dec( &my_counter );
```

The API also supports a number of other common use cases, including the operate-and-test routines. These allow the atomic variable to be manipulated and then tested (all performed as one atomic operation). One special function called `atomic_add_negative` is used to add to the atomic variable and then return true if the resulting value is negative. This is used by some of the architecture-dependent semaphore functions in the kernel.

While many of the functions do not return the value of the variable, two in particular operate and return the resulting value (`atomic_add_return` and `atomic_sub_return`), as shown in Listing 3.

Listing 3. Operate-and-test atomic functions

```
if (atomic_sub_and_test( 1, &my_counter )) {
  // my_counter is zero
}

if (atomic_dec_and_test( &my_counter )) {
  // my_counter is zero
}

if (atomic_inc_and_test( &my_counter )) {
  // my_counter is zero
}

if (atomic_add_negative( 1, &my_counter )) {
  // my_counter is less than zero
}

val = atomic_add_return( 1, &my_counter ));

val = atomic_sub_return( 1, &my_counter ));
```

If your architecture supports 64-bit long types (`BITS_PER_LONG` is 64), then `long_t atomic` operations are available. You can see the available long operations in linux/include/asm-generic/atomic.h.

You'll also find support for bitmask operations with the atomic API. Rather than arithmetic operations (as explored earlier), you'll find set and clear operations. Many drivers use these atomic operations, particularly SCSI. The use of bitmask atomic operations is slightly different than arithmetic because only two operations are available (set mask and clear mask). You provide a value and the bitmask upon which the operation is to be performed, as shown in Listing 4.

Listing 4. Bitmask atomic functions

```
unsigned long my_bitmask;

atomic_clear_mask( 0, &my_bitmask );

atomic_set_mask( (1<<24), &my_bitmask );
```

Atomic API prototypes
Implementations for the atomic operations are architecture dependent and can therefore be found in ./linux/include/asm-<arch>/atomic.h.

Spinlocks

Spinlocks are a special way of ensuring mutual exclusion using a busy-wait lock. When the lock is available, it is taken, the mutually-exclusive action is performed, and then the lock is released. If

the lock is not available, the thread busy-waits on the lock until it is available. While busy-waiting may seem inefficient, it can actually be faster than putting the thread to sleep and then waking it up later when the lock is available.

Spinlocks are really only useful in SMP systems, but because your code will end up running on an SMP system, adding them for UP systems is the right thing to do.

Spinlocks are available in two varieties: full locks and reader/writer locks. Let's look at the full lock variety first.

First you create a new spinlock through a simple declaration. This can be initialized in place or through a call to `spin_lock_init`. Each of the variants shown in Listing 5 accomplishes the same result.

Listing 5. Creating and initializing a spinlock

```
spinlock_t my_spinlock = SPIN_LOCK_UNLOCKED;

... or ...

DEFINE_SPINLOCK( my_spinlock );

... or ...

spin_lock_init( &my_spinlock );
```

Now that you have a spinlock defined, there are a number of locking variants that you can use. Each is useful in different contexts.

First is the `spin_lock` and `spin_unlock` variant shown in Listing 6. This is the simplest and performs no interrupt disabling but includes full memory barriers. This variant assumes no interactions with interrupt handlers and this lock.

Listing 6. Spinlock lock and unlock functions

```
spin_lock( &my_spinlock );

// critical section

spin_unlock( &my_spinlock );
```

Next is the `irqsave` and `irqrestore` pair shown in Listing 7. The `spin_lock_irqsave` function acquires the spinlock and disables interrupts on the local processor (in the SMP case). The `spin_unlock_irqrestore` function releases the spinlock and restores the interrupts (via the `flags` argument).

Listing 7. Spinlock variant with local CPU interrupt disable

```
spin_lock_irqsave( &my_spinlock, flags );

// critical section

spin_unlock_irqrestore( &my_spinlock, flags );
```

A less safe variant of `spin_lock_irqsave`/`spin_unlock_irqrestore` is `spin_lock_irq`/`spin_unlock_irq`. I recommend that you avoid this variant because it makes assumptions about interrupt states.

Finally, if your kernel thread shares data with a bottom half, then you can use another variant of the spinlock. A *bottom half* is a way to defer work from interrupt handling to be done later in a device driver. This version of the spinlock disables soft interrupts on the local CPU. This has the effect of preventing softirqs, tasklets, and bottom halves from running on the local CPU. This variant is shown in Listing 8.

Listing 8. Spinlock functions for bottom-half interactions

```
spin_lock_bh( &my_spinlock );

// critical section

spin_unlock_bh( &my_spinlock );
```

Reader/writer spinlocks

In many cases, access to data is indicated by many readers and less writers (accessing the data for read is more common than accessing for write). To support this model, reader/writer locks were created. What's interesting with this model is that multiple readers are permitted access to the data at one time, but only one writer. If a writer has the lock, no reader is allowed to enter the critical section. If only a reader has the lock, then multiple readers are permitted in the critical section. Listing 9 demonstrates this model.

Listing 9. Reader/writer spinlock functions

```
rwlock_t my_rwlock;

rwlock_init( &my_rwlock );

write_lock( &my_rwlock );

// critical section -- can read and write

write_unlock( &my_rwlock );


read_lock( &my_rwlock );

// critical section -- can read only

read_unlock( &my_rwlock );
```

You'll also find variants of reader/writer spinlocks for bottom halves and interrupt request (IRQ) saving depending on the situation for which you require the lock. Obviously, if your use of the lock is reader/writer in nature, this spinlock should be used over the standard spinlock, which doesn't differentiate between readers and writers.

Kernel mutexes

Mutexes are available in the kernel as a way to accomplish semaphore behavior. The kernel mutex is implemented on top of the atomic API, though this is not visible to the kernel user. The mutex

is simple, but there are some rules you should remember. Only one task may hold the mutex at a time, and only this task can unlock the mutex. There is no recursive locking or unlocking of mutexes, and mutexes may not be used within interrupt context. But mutexes are faster and more compact than the current kernel semaphore option, so if they fit your need, they're the choice to use.

You create and initialize a mutex in one operation through the `DEFINE_MUTEX` macro. This creates a new mutex and initializes the structure. You can see this implementation in ./linux/include/linux/mutex.h.

```
DEFINE_MUTEX( my_mutex );
```

The mutex API provides five functions: three are used for locking, one for unlocking, and another for testing a mutex. Let's first look at the locking functions. The first function, `mutex_trylock`, is used in situations where you want the lock immediately or you want control to be returned to you if the mutex is not available. This function is shown in Listing 10.

Listing 10. Trying to grab a mutex with `mutex_trylock`

```
ret = mutex_trylock( &my_mutex );
if (ret != 0) {
  // Got the lock!
} else {
  // Did not get the lock
}
```

If, instead, you want to wait for the lock, you can call `mutex_lock`. This call returns if the mutex is available; otherwise, it sleeps until the mutex is available. In either case, when control is returned, the caller holds the mutex. Finally, the `mutex_lock_interruptible` is used in situations where the caller may sleep. In this case, the function may return `-EINTR`. Both of these calls are shown in Listing 11.

Listing 11. Locking a mutex with the potential to sleep

```
mutex_lock( &my_mutex );

// Lock is now held by the caller.

if (mutex_lock_interruptible( &my_mutex ) != 0)  {

  // Interrupted by a signal, no mutex held

}
```

After a mutex is locked, it must be unlocked. This is accomplished with the `mutex_unlock` function. This function cannot be called from interrupt context. Finally, you can check the status of a mutex through a call to `mutex_is_locked`. This call actually compiles to an inline function. If the mutex is held (locked), then one is returned; otherwise, zero. Listing 12 demonstrates these functions.

Listing 12. Testing a mutex with `mutex_is_locked`

```
mutex_unlock( &my_mutex );

if (mutex_is_locked( &my_mutex ) == 0) {

  // Mutex is unlocked

}
```

While the mutex API has its constraints because it's based upon the atomic API, it's efficient and, therefore, worthwhile if it fits your need.

Big kernel lock

Finally, there remains the big kernel lock (BKL). Its use in the kernel is diminishing, but the remaining uses are the most difficult to remove. The BKL made multiprocessor Linux possible, but finer-grained locks have slowly replaced the BKL. The BKL is provided through the functions `lock_kernel` and `unlock_kernel`. See ./linux/lib/kernel_lock.c for more information.

Summary

Linux tends to be a Swiss Army knife when it comes to options, and its locking methods are no different. The atomic locks provide not only a locking mechanism but also arithmetic or bitwise operations simultaneously. Spinlocks offer a locking mechanism (mostly for SMP) and also reader/writer spinlocks that permit multiple readers but only a single writer to obtain a given lock. Finally, mutexes are a relatively new locking mechanism that provides a simple API built on top of atomics. Whatever you need, Linux has a locking scheme to protect your data.

Resources

Learn

- Read all of Tim's *Anatomy of...* articles on developerWorks.
- "Linux and symmetric multiprocessing" (developerWorks, March 2007) explores the ideas behind multiprocessing and developing applications for Linux that exploit SMP. Locking mechanisms have become even more important since the introduction of SMP.
- Rusty Russell's Unreliable Guide to Locking offers a slightly older discussion of Linux kernel locking.
- Read "The Big Kernel Lock lives on" on LWN.net to learn why it remains alive and kicking in the Linux kernel.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- At the Linux Kernel Archives, grab the latest Linux source. The Linux source itself is the most direct location for information about kernel operation. There's also quite a bit of documentation within the Documentation subdirectory (though some of it is quite dated).
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the author

M. Tim Jones

M. Tim Jones is an embedded software architect and the author of *GNU/Linux Application Programming*, *AI Application Programming*, and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.