KERNEL SYNCHRONIZATION

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license**.
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

2000-2020 Kaiwan N Billimoria
kaiwanTECH, Bangalore, India.

Table of Contents

Kernel Synchronization....4

What is Synchronization....4

The Need for Atomicity....5

Causes of concurrency in the kernel....10

Deadlocks....11

Deadlock Prevention....12

Important Guidelines....13

Spinlocks and Mutexes....16

Spin Locks Versus Mutexes....17

SIDEBAR - The Old Semaphore Interface....20

Semaphores Versus Mutexes....27

Specialized Locking....30

Atomic Operators....30

64-Bit Atomic Operations....31

Atomic Bit Operations....32

Reader-Writer Locks....35

Sequence Locks....36

Per-CPU Variables....37

RCU....42

Memory Barriers....43

Debugging....46

Appendix A :: Internal Implementation of Spinlock on the ARMv6 and above....53

Appendix B :: Internal Implementation of Spinlock on x86....56

CMPXCHG: Compare and Exchange....56

Kernel Synchronization

<< Much of the discussion below is from the participant text "Linux Kernel Development" 2nd / 3rd Ed by Robert Love, Novell Press. >>

What is Synchronization

- **Critical sections** are those regions of kernel code that must run *atomically* to prevent data corruption.
- It is considered a bug if it is possible for two or more threads to execute a critical section's code simultaneously.
- When this does happen it is called a "**race condition**".
- Guaranteeing that race conditions do not occour and preventing concurrency is called **synchronization.**

The Need for Atomicity

The Single Variable

Now, let's look at a specific computing example. Consider a simple shared resource, a single global integer, and a simple critical region, the operation of merely incrementing it:

```
i++;
```

This might translate into machine instructions to the computer's processor that resemble the following:

```
get the current value of i and copy it into a register
add one to the value stored in the register
write back to memory the new value of i
```

<<

An Experiment:

Take a small 'C' program:

```
static int g=41, dum;
static void foo(int anint)
{
      int ultr = anint*2;
      dum = ultr+16;
}

int main(int argc, char **argv)
{
      g ++;
      foo(g);
      exit(0);
}
```

Will it be safe? Atomic? - Yes, if it compiles to exactly ***one*** *machine language instruction only.*

Do see and experiment with https://godbolt.org/

Moral :: especially for the unoptimized cases, a simple 'g ++' can certainly become more than one assembly/machine language instructions and is hence unsafe to use without locking!

>>

Now, assume that there are two threads of execution, both enter this critical region, If both threads of execution read the initial value of i before it is incremented, both threads increment and save the same value. As a result, the variable i contains the value 8 when, in fact, it should now contain 9. This is one of the simplest examples of a critical region.
and the initial value of i is 7. The desired outcome is then similar to the following (with each row representing a unit of time):

Thread 1	*Thread 2*
get i (7)	—
increment i (7 -> 8)	—
write back i (8)	—
—	get i (8)
—	increment i (8 -> 9)
---	write back i (9)

As expected, 7 incremented twice is 9. A possible outcome, however, is the following:

Thread 1	*Thread 2*
get i (7)	get i (7)
increment i (7 -> 8)	—
—	increment i (7 -> 8)
write back i (8)	—
—	write back i (8)

(Above) Source: LKD3

… "The classic pedagogical example is your checking account. If two people are each independently trying to cash checks for $100 that you wrote when your account contains just $150, somehow the action of reading your balance to check for sufficient funds and

> then modifying your balance to withdraw the money has to be indivisible. Otherwise person A could check for sufficient funds, then person B could check, then person A gets their money, then person B, even though your account doesn't actually have of enough cash to cover both checks.” …
>
> *(Above) Source*

Causes of concurrency in the kernel

1. Hardware Interrupts – asynchronous interrupts will interrupt the currently executing thread at virtually any time.

2. A task in the kernel sleeps; this essentially invokes the scheduler (directly or indirectly), which schedules another thread to run.

3. SMP – n threads could execute truly simultaneously on n CPUs on the same kernel code/data.

4. From the 2.6 kernel on, kernel preemption is possible – one thread can preempt another.

What the kernel developer has to realize and take care of is:

- It is considered a bug if an interrupt occours in critical section code and the interrupt handler (top or bottom half) executes code that operates on the same data.

- It is inviting race conditions if kernel code sleeps in a critical section.

- It is again a dangerous race condition if on an SMP system (which a kernel developer must always assume she is writing code for anyway), two or more threads can execute the same critical section kernel code.

- It is considered a bug if critical section kernel code can be preempted by another kernel task.

Knowing the above, a kernel developer can and must identify critical sections of code and take measures to protect against race conditions and data corruption. The difficulty lies not in the actual locking of code/data, **it lies in the *correct and complete identification* of critical sections.**

Kernel-space – what needs protection?

Essentially, any and all **global and static data** in a piece of kernel code, that has any possibility of concurrent access, requires explicit protection.
Actually, it can be more than that: the list includes shared registers, shared memory, message queues/mailboxes of any sort.

Deadlocks

A *deadlock* is a condition involving one or more threads of execution and one or more resources, such that each thread is waiting for one of the resources, but all the resources are already held. The threads are all waiting for each other, *but they will never make any progress toward releasing the resources that they already hold*. Therefore, none of the threads can continue, which means we have a deadlock.

A good analogy is a four-way traffic stop. If each car at the stop decides to wait for the other cars before going, no car will ever go and we have a traffic deadlock.

The simplest example of a deadlock is the self-deadlock[4]: If a thread of execution attempts to acquire a lock it already holds, it has to wait for the lock to be released.

[4] Some kernels prevent this type of deadlock by having recursive locks. These are locks that a single thread of execution may acquire multiple times. Linux, thankfully, does not provide recursive locks. This is usually considered a good thing. Although recursive locks might alleviate the self-deadlock problem, they very readily lead to sloppy locking semantics.

But it will never release the lock, because it is busy waiting for the lock, and the result is deadlock:

```
acquire lock
(attempt to) acquire lock, again
wait for lock to become available
...
```

Similarly, consider n threads and n locks. If each thread holds a lock that the other thread wants, all threads block while waiting for their respective locks to become available. The most common example is with two threads and two locks, which is often called the deadly embrace or the ABBA deadlock:

```
Thread 1                        Thread 2
...                             ...
acquire lock A                  acquire lock B
...                             ...
try to acquire lock B     try to acquire lock A
waits for lock B          waits for lock A
```

Each thread is waiting for the other and neither thread will ever release its original lock; therefore, neither lock will ever become available; the result: deadlock.

Deadlock Prevention

Prevention of deadlock scenarios is important. Although it is difficult to prove that code is free of deadlocks, it is possible to write deadlock-free code. A few simple rules go a long way:

- ***Lock ordering is vital. Nested locks must always be obtained in the same order.*** This prevents the deadly embrace deadlock. *Document* the lock ordering so others will follow it.

- Prevent starvation. Ask, does this code always finish? If foo does not occur, will bar wait forever?

- Do not double acquire the same lock.

- Complexity in your locking scheme invites deadlocks, so design for simplicity.

The first point is important, and worth stressing. If two or more locks are ever acquired at the same time, they must always be acquired in the same order. Let's assume you have the cat, dog, and fox lock that protect data structures of the same name. Now assume you have a function that needs to work on all three of these data structures simultaneously - perhaps to copy data between them. Whatever the case, the data structures require locking to ensure safe access. *If one function acquires the locks in the order cat, dog, and then fox, then every other function must obtain these locks (or a subset of them) in this same order*. For example, it is a potential deadlock (and hence a bug) to first obtain the fox lock, and then obtain the dog lock (because the dog lock must always be acquired prior to the fox lock).

--snip--

<<
Some examples of following the "aquire locks in the same order" rule in the Linux

kernel: (note, we're not seeing the code, just the ***important*** *fact that it's clearly documented) (ver 2.6.23):*

Eg. 1:
kernel/sched.c:

```
--snip--
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
       spinlock_t lock;        /* runqueue lock */
       ...
--snip--
```

Eg. 2: virt/kvm/kvm_main.c

```
...
MODULE_AUTHOR("Qumranet");
MODULE_LICENSE("GPL");

/*
 * Ordering of locks:
 *
 *      kvm->lock --> kvm->slots_lock --> kvm->irq_lock
 */
...
```

Eg. 3: mm/slub.c

```
...
 * Lock order:
 *   1. slab_mutex (Global Mutex)
 *   2. node->list_lock
 *   3. slab_lock(page) (Only on some arches and for debugging)
...
>>
```

Important Guidelines

- **Lock *data*, not code.**
- Refine your locks' *granularity* (coarse vs. fine-grained) evolving toward finer-grained locks as these help reduce bottlenecks that typically occour around a (especially highly-*contended*) lock.
- Start simple and grow in complexity only as needed. Simplicity is key.
- Making your code SMP-safe is not something that can be added as an afterthought. Proper synchronization - locking that is free of deadlocks, scalable, and clean - requires design decisions from start through finish. Whenever you write kernel code, whether it is a new system call or a rewritten driver, protecting data from concurrent access needs to be a primary concern. Provide sufficient protection for every scenario - SMP, kernel preemption, and so on - and rest assured the data will be safe on any given machine and configuration.

<<

- We understand that we *require locking* for concurrent code paths that work on any kind of shared data
- However, it's really important to also understand that *locking creates bottlenecks*
- Good physical analogies, perhaps:
  - a funnel; the stem of the funnel is the critical section – it's only wide enough to allow one thread at a time
  - a *single* toll booth on a busy highway

Image Source
>>

Concurrency in the Kernel

Much of the material below is from the text "Essential Linux Device Drivers" by Sreekrishnan Venkateshwaran, published by Prentice Hall (details below).
"Essential Linux Device Drivers"
Hardcover: 744 pages

- Author: Sreekrishnan Venkateswaran
- Publisher: Prentice Hall PTR Open Source Software Development Series; 1 edition (April 6, 2008)
- Language: English
- ISBN-10: 0132396556
- ISBN-13: 978-0132396554

This book is "Safari-enabled" and has been legally purchased. It's soft-copy form is used here solely as a training aid (adhering to the Open Publication License that this book is released under, see details below). To gain access to this material soft-copy, please purchase and register an account with http://safari.informit.com/ or purchase the (hard-copy) book, whcih grants a 45-day Safari account for this book. All copyrights for this material reserved by the author and publisher.

The book's licensing policy states the following: "This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/)."

You might find additional notes (and/or examples, source code, program output, images, etc), that are appended as a teaching aid; these are << generally displayed within angle brackets as shown here. >>

With the arrival of multicore laptops, Symmetric Multi Processing (SMP) is no longer confined to the realm of hi-tech users. SMP and kernel preemption are scenarios that generate multiple threads of execution. These threads can simultaneously operate on shared kernel data structures. Because of this, accesses to such data structures have to be serialized.

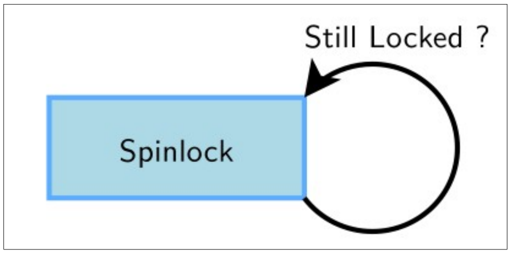
Let's discuss the basics of protecting shared kernel resources from concurrent access. We start with a simple example and gradually introduce complexities such as interrupts, kernel preemption, and SMP.

Spinlocks and Mutexes

<< Instructor Note: Mention RML's rooms-with-people analogy... >>

A code area that accesses shared resources is called a critical section. Spinlocks and mutexes (short for mutual exclusion) are the two basic mechanisms used to protect critical sections in the kernel. Let's look at each in turn.

A spinlock ensures that only a single thread enters a critical section at a time. Any other thread that desires to enter the critical section has to remain spinning at the door until the first thread exits. Note that we use the term thread to refer to a thread of execution, rather than a kernel thread.



The basic spinlock usage is as follows:

```
#include <linux/spinlock.h>
spinlock_t mylock;

spin_lock_init (&mylock);  /* Initialize (to unlocked state) */

/* or use the macro:
  DEFINE_SPINLOCK (mylock);
  to declare and initialize (to the unlocked state) a spinlock.
*/

/* Acquire the spinlock. This is inexpensive if there
 * is no one inside the critical section. In the face of
 * contention, spinlock() has to busy-wait.
 */
spin_lock(&mylock);

/* ... Critical Section code ... */

spin_unlock(&mylock); /* Release the lock */
```

<<

- **Larger projects**
  - (especially early in development) be careful not to keep too few locks; performance suffers!
  - When there are hundreds and thousands of locks, *naming* the lock variable can become a headache
    - Kernel devs use the technique of making the lock variable a member of the data structure it protects!
    - Eg. in the task structure, *alloc_lock* is a member of the structure:

```
        ...
        /* Protection against (de-)allocation: mm, files, fs, tty,
        keyrings, mems_allowed, mempolicy: */
        spinlock_t          alloc_lock;
           ...
```

- in the *file* structure:

```
struct file {
    union {
 ...
    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t      f_lock;
    enum rw_hint    f_write_hint;
...
```

  - in the *block_device* structure:

```
struct block_device {
   dev_t           bd_dev;  /* not a kdev_t - it's a search key */
   int         bd_openers;
...
   struct mutex        bd_mutex;   /* open/close mutex */
...
```

  - and so on...

>>

In contrast to spinlocks that put threads into a spin if they attempt to enter a busy critical section, mutexes put contending threads to sleep until it's their turn to occupy the critical section (by calling `schedule_preempt_disabled()`). Because it's a bad thing to consume processor cycles to spin, mutexes are more suitable than spinlocks to protect critical sections when the estimated wait time is long. In mutex terms, anything more than two context switches is considered long, because a mutex has to switch out the contending thread to sleep, and switch it back in when it's time to wake it up.

Spin Locks Versus Mutexes

In many cases, therefore, it's easy to decide whether to use a spinlock or a mutex:

- If the critical section needs to sleep, you have no choice but to use a mutex. It's illegal to schedule, preempt, or sleep on a wait queue (or any other way) after acquiring a spinlock.

- Because mutexes put the calling thread to sleep in the face of contention, you have no choice but to use spinlocks inside interrupt handlers (or any kind of softirq context).

Knowing when to use a spin lock versus a mutex (or semaphore) is important to writing optimal code. In many cases, however, there is little choice. Only a spin lock can be used in interrupt context, whereas only a mutex can be held while a task sleeps. The table below reviews the requirements that dictate which lock to use.

What to Use: Spin Locks Versus Semaphores

Requirement	*Recommended Lock*
Low overhead locking	Spin lock is preferred.
Short lock hold time	Spin lock is preferred.
Long lock hold time	Mutex is preferred.
Need to lock from interrupt context	Spin lock is required.
Need to sleep while holding lock	Mutex is required.

Basic mutex usage is as follows:

```
#include <linux/mutex.h>

/* Statically declare a mutex. To dynamically
   create a mutex, use mutex_init() */
static DEFINE_MUTEX(mymutex);

/* Acquire the mutex. This is inexpensive if there
 * is no one inside the critical section. In the face of
 * contention, mutex_lock() puts the calling thread to sleep.
 */
mutex_lock(&mymutex);

/* ... Critical Section code ... */

mutex_unlock(&mymutex);      /* Release the mutex */
```

Also, note that mutex_lock() puts the caller process context into an *uninterruptible sleep;* this is usually not required. The recommendation is to, as far as possible, use the

`mutex_lock_interruptible(struct mutex *lock)`

variant, which puts the caller process context into an *interruptible sleep*.

SIDEBAR

A FAQ is: is there an *ordering* to how a mutex unlock is performed?

Background Info:

Visualize a scenario where several threads attempt to take a mutex lock more or less simultaneously. Of course, only one of them will aquire it (becoming the “winner” or “owner” of the mutex); the other “loser threads” will wait upon the unlock event by sleeping. Yes, that's fine.

But when the owner performs the unlock, *which* of the waiting threads will aquire the mutex lock becoming the next winner/owner?

The documentation states that all things being equal (peers as far as priorities, scheduling policy, etc is concerned), the selection of the new winner thread is arbitrary or random and we should not depend on a certain thread aquiring the lock.

Well, that's not the complete truth: in reality it is architecture dependant.
The x86 port of Linux (on the mutex-aquire 'slowpath'), *queues* the “loser” threads in a FIFO fashion on what's essentially a linked list; threads desiring to take the mutex lock are queued onto this list in a FIFO fashion.

In *kernel/mutex.c* :

…

```
/*
 * Lock a mutex (possibly interruptible), slowpath:
 */
static inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int
subclass,
                    struct lockdep_map *nest_lock, unsigned long ip)
{
        struct task_struct *task = current;
        struct mutex_waiter waiter;
        unsigned long flags;

...
 /* add waiting tasks to the end of the waitqueue (FIFO): */
        list_add_tail(&waiter.list, &lock->wait_list);
        waiter.task = task
...
```

SIDEBAR - The Old Semaphore Interface

The mutex interface, which replaces the older semaphore interface, originated in the -rt tree and was merged into the mainline with the 2.6.16 kernel release. The semaphore interface is still around, however. Basic usage of the semaphore interface is as follows:

```
#include <asm/semaphore.h>  /* Architecture dependent header */

/* Statically declare a semaphore. To dynamically
   create a semaphore, use init_MUTEX() */
static DECLARE_MUTEX(mysem);

down(&mysem);    /* Acquire the semaphore */

/* ... Critical Section code ... */
up(&mysem);      /* Release the semaphore */
```

(Counting) Semaphores can be configured to allow a predetermined number of threads into the critical section simultaneously. However, semaphores that permit more than a single holder at a time are rarely used (i.e. binary semaphores are the norm).

To illustrate the use of concurrency protection, let's start with a critical section (keep in mind that this entire discussion is for a critical section in *kernel* space) that is present only in process context and gradually introduce complexities in the following order:

1. Critical section present only in process context on a Uniprocessor (UP) box running a nonpreemptible kernel.
2. Critical section present in process and interrupt contexts on a UP machine running a nonpreemptible kernel.
3. Critical section present in process and interrupt contexts on a UP machine running a preemptible kernel.
4. Critical section present in process and interrupt contexts on an SMP machine running a preemptible kernel.

Case 1: Process Context, UP Machine, No Preemption

This is the simplest case and needs no locking, so we won't discuss this further.

Case 2: Process and Interrupt Contexts, UP Machine, No Preemption

In this case, you need to only disable interrupts to protect the critical region. To see why, assume that A and B are process context threads, and C is an interrupt context thread, all vying to enter the same critical section.

Because Thread C is executing in interrupt context and always runs to completion before yielding to Thread A or Thread B, it need not worry about protection. Thread A, for its part, need not be concerned about Thread B (and vice versa) because the kernel is not preemptible. Thus, Thread A and Thread B need to guard against only the possibility of Thread C stomping through the critical section while they are inside the same section. They achieve this by disabling interrupts prior to entering the critical section:

```
Point A:
  local_irq_disable();  /* Disable Interrupts in local CPU */
  /* ... Critical Section ...  */
  local_irq_enable();   /* Enable Interrupts in local CPU */
```

<<

Alternatively, to get the same behaviour as above using the spinlock API:

```
Point A:
  spin_lock_irq(&mylock); /* disable local irq's & aquire the lock */
  /* ... Critical Section code ... */
  spin_unlock_irq(&mylock); /* release the lock & enable local
irq's */
>>
```

However, if interrupts were already disabled when execution reached Point A, local_irq_enable() creates the unpleasant side effect of reenabling interrupts, rather than restoring interrupt state. This can be fixed as follows:

```
unsigned long flags;

Point A:
  local_irq_save(flags);     /* Disable Interrupts */
  /* ... Critical Section ... */
  local_irq_restore(flags);  /* Restore state to what
                                it was at Point A */
```

This works correctly irrespective of the interrupt state at Point A.

<<

Alternatively, to get the same behaviour as above using the spinlock API:

```
Point A:
  spin_lock_irqsave(&mylock, flags); /* save state, disable local
                                        irq's & aquire the lock */
  /* ... Critical Section code ... */
  spin_unlock_irq_restore(&mylock, flags); /* release the lock, &
                                              restore state  */
```

>>

[*Documentation/spinlocks.txt*]() - Linus

--snip--

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).
> *--snip--*

Case 3: Process and Interrupt Contexts, UP Machine, Preemption

If preemption is enabled, mere disabling of interrupts won't protect your critical region from being trampled over. There is the possibility of multiple threads simultaneously entering the critical section in process context. Referring back to Figure 2.4 in this scenario, Thread A and Thread B now need to protect themselves against each other in addition to guarding against Thread C. The solution apparently, is to disable kernel preemption before the start of the critical section and reenable it at the end, in addition to disabling/reenabling interrupts. For this, Thread A and Thread B use the irq variant of spinlocks:

```
unsigned long flags;

Point A:
  /* Save interrupt state.
   * Disable interrupts - this implicitly disables preemption */
  spin_lock_irqsave(&mylock, flags);

  /* ... Critical Section ... */

  /* Restore interrupt state to what it was at Point A */
  spin_unlock_irqrestore(&mylock, flags);
```

Preemption state need not be explicitly restored to what it was at Point A because the kernel internally does that for you via a variable called the preemption counter. The counter gets incremented whenever preemption is disabled (using preempt_disable()) and gets decremented whenever preemption is enabled (using preempt_enable()). Preemption kicks in only if the counter value is zero.

Case 4: Process and Interrupt Contexts, SMP Machine, Preemption

Let's now assume that the critical section executes on an SMP machine. Your kernel has been configured with CONFIG_SMP and CONFIG_PREEMPT turned on.

In the scenarios discussed this far, spinlock primitives have done little more than enable/disable preemption and interrupts. The actual locking functionality has been compiled away. In the presence of SMP, the locking logic gets compiled in, and the spinlock primitives are rendered SMP-safe. The SMP-enabled semantics is as follows:

```
unsigned long flags;

Point A:
  /*
    - Save interrupt state on the local CPU
    - Disable interrupts on the local CPU. This implicitly disables
      preemption.
    - Lock the section to regulate access by other CPUs
   */
  spin_lock_irqsave(&mylock, flags);

  /* ... Critical Section ... */

  /*
    - Restore interrupt state and preemption to what it
      was at Point A for the local CPU
    - Release the lock
   */
  spin_unlock_irqrestore(&mylock, flags);
```

On SMP systems, only interrupts on the local CPU are disabled when a spinlock is acquired. So, a process context thread (say Thread A in Figure 2.4) might be running on one CPU, while an interrupt handler (say Thread C in Figure 2.4) is executing on another CPU. An interrupt handler on a nonlocal processor thus needs to spin-wait until the process context code on the local processor exits the critical section. The interrupt context code calls spin_lock()/spin_unlock() to do this:

```
spin_lock(&mylock);

/* ... Critical Section ... */

spin_unlock(&mylock);
```

Documentation/spinlocks.txt **- Linus**

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.
>
> *--snip--*
>
> 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.
>
> *--snip--*

<<
An example of using *spin_[un|lock]* can be seen in the RealTek 8139C+ network driver's *(drivers/net/ethernet/realtek/8139cp.c) interrupt handler routine (ISR)*:

```
...
static irqreturn_t cp_interrupt (int irq, void *dev_instance)
{
	struct net_device *dev = dev_instance;
	struct cp_private *cp;
...
	cpw16(IntrStatus, status & ~cp_rx_intr_mask);

	spin_lock(&cp->lock);

	/* close possible race's with dev_close */
	if (unlikely(!netif_running(dev))) {
```

```
                cpw16(IntrMask, 0);
                spin_unlock(&cp->lock);
                return IRQ_HANDLED;
        }

        if (status & (RxOK | RxErr | RxEmpty | RxFIFOOvr))
                if (napi_schedule_prep(&cp->napi)) {
                        cpw16_f(IntrMask, cp_norx_intr_mask);
                        __napi_schedule(&cp->napi);
                }

        if (status & (TxOK | TxErr | TxEmpty | SWInt))
                cp_tx(cp);
        if (status & LinkChg)
                mii_check_media(&cp->mii_if, netif_msg_link(cp), false);

        spin_unlock(&cp->lock);
…
}
```

and in the poll routine

```
...
        spin_lock_irqsave(&cp->lock, flags);
        __napi_complete(napi);
        cpw16_f(IntrMask, cp_intr_mask);
        spin_unlock_irqrestore(&cp->lock, flags);
...
```

>>

void spin_lock_bh(spinlock_t *lock) :
This variant is used to also disable execution of all bottom halves (until the spin_unlock_bh() kicks in).

<<
Tips

- ***spin_lock*** : *it too, down the line,* ***disables interrupts & preemption!***

```
include/linux/spinlock.h:spin_lock
   raw_spin_lock
      kernel/locking/spinlock.c:_raw_spin_lock
         include/linux/spinlock_api_smp.h:__raw_spin_lock
             include/linux/lockdep.h:spin_aquire
                 lock_acquire_exclusive
                     kernel/locking/lockdep.c:lock_aquire()
/*
* We are not always called with irqs disabled - do that here,
* and also avoid lockdep recursion:
```

```
*/
```

- Browse through the RealTek RTL-8139 network driver (*drivers/net/ethernet/realtek/8139cp.c*) to see several examples of spinlock usage.
- A brief interesting explanation on the implementation of spinlocks on ARM processors.
- Internal implementation of spinlocks on the x86 (possibly an old implementation).

>>

Semaphores Versus Mutexes

Mutexes and semaphores are similar. Having both in the kernel is confusing. Thankfully, the formula dictating which to use is quite simple: Unless one of mutex's additional constraints prevent you from using them, prefer the new mutex type to semaphores. When writing new code, only specific, often low-level, uses need a semaphore. Start with a mutex and move to a semaphore only if you run into one of their constraints and have no other alternative.

SIDEBAR

A good example of seeing how the Linux kernel mainline tree is maintained (in this case, older or deprecated interfaces being removed in favour of the newer ones): the commit below shows how the older 'semaphore' kernel API/interfaces have been replaced by the newer 'mutex' interfaces (example below is on the lm70 temperature sensor device driver).

URL: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=4bfe66048e97d29ab229519e9a821dbd4d929bd9

/pub/scm / linux/kernel/git/torvalds/linux-2.6.git / commitdiff

commit ? search:

summary | shortlog | log | commit | commitdiff | tree
raw (parent: 8de5770)

```
hwmon: (lm70) Convert semaphore to mutex
Matthias Kaehlcke [Wed, 24 Oct 2007 12:59:09 +0000 (14:59 +0200)]
Signed-off-by: Matthias Kaehlcke <matthias.kaehlcke@gmail.com>
Acked-by: Jean Delvare <khali@linux-fr.org>
Signed-off-by: Mark M. Hoffman <mhoffman@lightlink.com>


 drivers/hwmon/lm70.c             patch | blob | history

diff --git a/drivers/hwmon/lm70.c b/drivers/hwmon/lm70.c
index dd36688..d435f00 100644 (file)
--- a/drivers/hwmon/lm70.c
+++ b/drivers/hwmon/lm70.c
@@ -31,14 +31,15 @@
 #include <linux/err.h>
 #include <linux/sysfs.h>
 #include <linux/hwmon.h>
+#include <linux/mutex.h>
 #include <linux/spi/spi.h>
-#include <asm/semaphore.h>
+

 #define DRVNAME                 "lm70"

 struct lm70 {
         struct device *hwmon_dev;
-        struct semaphore sem;
+        struct mutex lock;
 };

 /* sysfs hook function */
@@ -51,7 +52,7 @@ static ssize_t lm70_sense_temp(struct device *dev,
         s16 raw=0;
         struct lm70 *p_lm70 = dev_get_drvdata(&spi->dev);

-        if (down_interruptible(&p_lm70->sem))
+        if (mutex_lock_interruptible(&p_lm70->lock))
                 return -ERESTARTSYS;

         /*
@@ -83,7 +84,7 @@ static ssize_t lm70_sense_temp(struct device *dev,
         val = ((int)raw/32) * 250;
         status = sprintf(buf, "%d\n", val); /* milliddegrees Celsius */
 out:
-        up(&p_lm70->sem);
+        mutex_unlock(&p_lm70->lock);
         return status;
 }

@@ -112,7 +113,7 @@ static int __devinit lm70_probe(struct spi_device *spi)
         if (!p_lm70)
                 return -ENOMEM;

-        init_MUTEX(&p_lm70->sem);
+        mutex_init(&p_lm70->lock);

         /* sysfs hook */
```

```
        p_lm70->hwmon_dev = hwmon_device_register(&spi->dev);
Linus' kernel tree                                           RSS Atom
```

Spin Locks Versus Mutexes

Knowing when to use a spin lock versus a mutex (or semaphore) is important to writing optimal code. In many cases, however, there is little choice. Only a spin lock can be used in interrupt context, whereas only a mutex can be held while a task sleeps. Table 10.8 reviews the requirements that dictate which lock to use.

What to Use: Spin Locks Versus Semaphores

Requirement	*Recommended Lock*
Low overhead locking	Spin lock is preferred.
Short lock hold time	Spin lock is preferred.
Long lock hold time	Mutex is preferred.
Need to lock from interrupt context	Spin lock is required.
Need to sleep while holding lock	Mutex is required.

Specialized Locking

The kernel has specialized locking primitives in its repertoire that help improve performance under specific conditions. Using a mutual-exclusion scheme tailored to your needs makes your code more powerful. Let's take a look at some of the specialized exclusion mechanisms.

Atomic Operators

Atomic operators are used to perform lightweight one-shot operations such as bumping counters, conditional increments, and setting bit positions. Atomic operations are guaranteed to be serialized and do not need locks for protection against concurrent access. The implementation of atomic operators is architecture-dependent.

You must declare your integer variable as atomic_t *and initialize it with* ATOMIC_INIT().

<<
Source: "Linux Kernel Development" by Robert Love, 2nd Ed., Novell Press >>

Table : Full Listing of Atomic Integer Operations

Atomic Integer Operation	**Description**
`ATOMIC_INIT(int I)`	At declaration, initialize an atomic_t to I
`int atomic_read(atomic_t *v)`	Atomically read the integer value of v
`void atomic_set(atomic_t *v, int I)`	Atomically set v equal to I
`void atomic_add(int i, atomic_t *v)`	Atomically add i to v
`void atomic_sub(int i, atomic_t *v)`	Atomically subtract i from v
`void atomic_inc(atomic_t *v)`	Atomically add one to v
`void atomic_dec(atomic_t *v)`	Atomically subtract one from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Atomically subtract i from v and return true if the result is zero; otherwise false
`int atomic_add_negative(int i, atomic_t *v)`	Atomically add i to v and return true if the result is negative; otherwise false
`int atomic_dec_and_test(atomic_t *v)`	Atomically decrement v by one and return true if zero; false otherwise
`int atomic_inc_and_test(atomic_t *v)`	Atomically increment v by one and return true if the result is zero; false otherwise

>>

Atomic operations eliminate the hassle of using locks to protect a single integer variable from concurrent access.

64-Bit Atomic Operations

With the rising prevalence of 64-bit architectures, it is no surprise that the Linux kernel developers augmented the 32-bit atomic_t type with a 64-bit variant, atomic64_t. For portability, the size of atomic_t cannot change between architectures, so atomic_t is 32-bit even on 64-bit architectures.

Instead, the atomic64_t type provides a 64-bit atomic integer that functions otherwise identical to its 32-bit brother. Usage is exactly the same, except that the usable range of the integer is 64, rather than 32, bits. Nearly all the classic 32-bit atomic operations are implemented in 64-bit variants; they are prefixed with atomic64 in lieu of atomic.

Table 10.2 is a full listing of the standard operations; some architectures implement more, but they are not portable. As with atomic_t, the atomic64_t type is just a simple wrapper around an integer, this type a long:

```
typedef struct {
   volatile long counter;
} atomic64_t;
```

Table 10.2

Atomic Integer Methods

Atomic Integer Operation	***Description***
ATOMIC64_INIT(long i)	At declaration, initialize to i.
long atomic64_read(atomic64_t *v)	Atomically read the integer value of v.
void atomic64_set(atomic64_t *v, int i)	Atomically set v equal to i.
void atomic64_add(int i, atomic64_t *v)	Atomically add i to v.
void atomic64_sub(int i, atomic64_t *v)	Atomically subtract i from v.
void atomic64_inc(atomic64_t *v)	Atomically add one to v.
void atomic64_dec(atomic64_t *v)	Atomically subtract one from v.
int atomic64_sub_and_test(int i, atomic64_t *v)	Atomically subtract i from v and return true if the result is zero;

	otherwise false.
int atomic64_add_negative(int i, atomic64_t *v)	Atomically add i to v and return true if the result is negative; otherwise false.
long atomic64_add_return(int i, atomic64_t *v)	Atomically add i to v and return the result.
long atomic64_sub_return(int i, atomic64_t *v)	Atomically subtract i from v and return the result.
long atomic64_inc_return(int i, atomic64_t *v)	Atomically increment v by one and return the result.
long atomic64_dec_return(int i, atomic64_t *v)	Atomically decrement v by one and return the result.
int atomic64_dec_and_test(atomic64_t *v)	Atomically decrement v by one and return true if zero; false otherwise.
int atomic64_inc_and_test(atomic64_t *v)	Atomically increment v by one and return true if the result is zero; false otherwise.

NOTE:-

Due to security concerns (with IoF: integer overflow / underflows), `atomic_t` is being slowly converted to `refcount_t` in recent kernels.

Atomic Bit Operations

The kernel also supports operators such as set_bit(), clear_bit(), and test_and_set_bit() to atomically engage in bit manipulations.

<<

Why?
In a driver, when performing an operation on a register (often, set/clear a bit), it is important to use this sequence:

Read
Modify

Write

>>

Look at include/asm-<your-arch>/atomic.h for the atomic operators supported on your architecture.

<<
Source: "Linux Kernel Development" by Robert Love, 2nd Ed., Novell Press >>

Table : Listing of Atomic Bitwise Operations

Atomic Bitwise Operation	**Description**
`void set_bit(int nr, void *addr)`	Atomically set the nr-th bit starting from addr
`void clear_bit(int nr, void *addr)`	Atomically clear the nr-th bit starting from addr
`void change_bit(int nr, void *addr)`	Atomically flip the value of the nr-th bit starting from addr
`int test_and_set_bit(int nr, void *addr)`	Atomically set the nr-th bit starting from addr and return the previous value
`int test_and_clear_bit(int nr, void *addr)`	Atomically clear the nr-th bit starting from addr and return the previous value
`int test_and_change_bit(int nr, void *addr)`	Atomically flip the nr-th bit starting from addr and return the previous value
`int test_bit(int nr, void *addr)`	Atomically return the value of the nr-th bit starting from addr

>>

Because the functions operate on a generic pointer, there is no equivalent of the atomic integer's atomic_t type. Instead, you can work with a pointer to whatever data you want. Consider an example:

```
unsigned long word = 0;

set_bit(0, &word);      /* bit zero is now set (atomically) */
set_bit(1, &word);      /* bit one is now set (atomically) */
printk("%ul\n", word);  /* will print "3" */
clear_bit(1, &word);    /* bit one is now unset (atomically) */
change_bit(0, &word);   /* bit zero is flipped; now it is unset
                           (atomically) */

/* atomically sets bit zero and returns the previous value (zero) */
if (test_and_set_bit(0, &word)) {
/* never true ... */
}

/* the following is legal; you can mix atomic bit instructions with normal C */
word = 7;
```

Conveniently, nonatomic versions of all the bitwise functions are also provided. They behave identically to their atomic siblings, except they do not guarantee atomicity, and their names are prefixed with double underscores. For example, the nonatomic form of test_bit() is __test_bit(). If you do not require atomicity (say, for example, because a lock already protects your data), these variants of the bitwise functions might be faster.

Reader-Writer Locks

Another specialized concurrency regulation mechanism is a reader-writer variant of spinlocks. If the usage of a critical section is such that separate threads either read from or write to a shared data structure, but don't do both, these locks are a natural fit. Multiple reader threads are allowed inside a critical region simultaneously. Reader spinlocks are defined as follows:

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;

read_lock(&myrwlock);     /* Acquire reader lock */
/* ... Critical Region ... */
read_unlock(&myrwlock);   /* Release lock */
```

However, if a writer thread enters a critical section, other reader or writer threads are not allowed inside. To use writer spinlocks, you would write this:

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;

write_lock(&myrwlock);    /* Acquire writer lock */
/* ... Critical Region ... */
write_unlock(&myrwlock);  /* Release lock */
```

Look at the IPX routing code present in *net/ipx/ipx_route.c* for a real-life example of a reader-writer spinlock. A reader-writer lock called *ipx_routes_lock* protects the IPX routing table from simultaneous access. Threads that need to look up the routing table to forward packets request reader locks. Threads that need to add or delete entries from the routing table acquire writer locks. This improves performance because there are usually far more instances of routing table lookups than routing table updates.

Like regular spinlocks, reader-writer locks also have corresponding irq variants - namely, `read_lock_irqsave()`, `read_lock_irqrestore()`, `write_lock_irqsave()`, and `write_lock_irqrestore()`. The semantics of these functions are similar to those of regular spinlocks.

Source : *Documentation/spinlocks.txt* – Linus
--snip--

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.

--snip--

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.txt* for complete information.)

<<
Also, from LDD3:
“Reader/writer locks can starve readers just as rwsems can. This behavior is rarely a problem; however, if there is enough lock contention to bring about starvation, performance is poor anyway.”
>>

Sequence Locks

Sequence locks or seqlocks, introduced in the 2.6 kernel, are reader-writer locks where writers are favored over readers. This is useful if write operations on a variable far outnumber read accesses. An example is the jiffies_64 variable discussed earlier in this chapter. Writer threads do not wait for readers who may be inside a critical section. Because of this, reader threads may discover that their entry inside a critical section has failed and may need to retry:

```
u64 get_jiffies_64(void) /* Defined in kernel/time.c */
{
  unsigned long seq;
  u64 ret;
  do {
     seq = read_seqbegin(&xtime_lock);
     ret = jiffies_64;
  } while (read_seqretry(&xtime_lock, seq));
  return ret;
}
```

Writers protect critical regions using `write_seqlock()` and `write_sequnlock()`.

Per-CPU Variables

Source: Per-CPU variables and the realtime tree [LWN, Jon Corbet]

--snip--

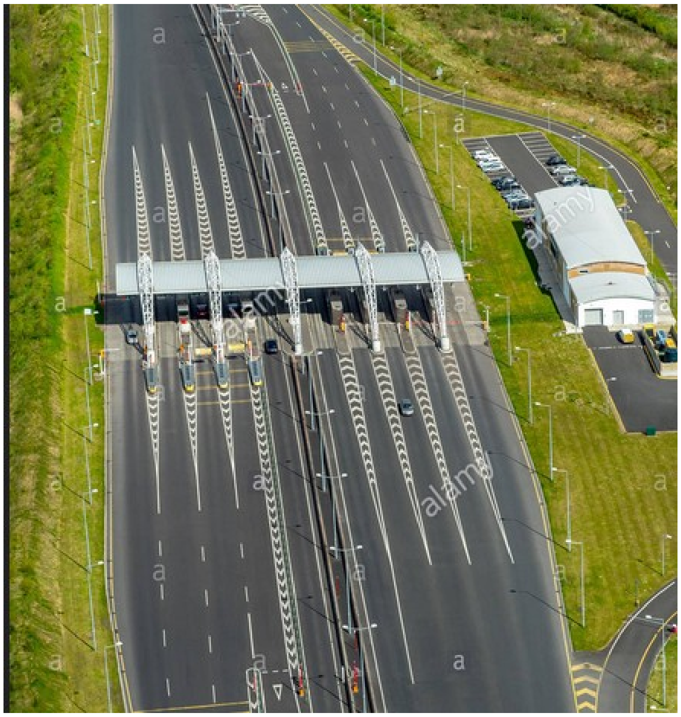
Symmetric multiprocessing systems are nice in that they offer equal access to memory from all CPUs. But taking advantage of the feature is a guaranteed way to create a slow system. Shared data requires mutual exclusion to avoid concurrent access; that means locking and the associated bottlenecks.

<<

Image Source

>>

Even in the absence of lock contention, simply moving cache lines between CPUs can wreck performance. The key to performance on SMP systems is minimizing the sharing of data, so it is not surprising that a great deal of scalability work in the kernel depends on the use of per-CPU data.



A per-CPU variable in the Linux kernel is actually an array with one instance of the variable for each processor. Each processor works with its own copy of the variable; this can be done with no locking, and with no worries about cache line bouncing.

<< *Image Source* >>

For example, some slab allocators maintain per-CPU lists of free objects and/or pages; these allow quick allocation and deallocation without the need for locking to exclude any other CPUs << also realize that the slab allocator API (*kmalloc* & friends) generally boils down to the Buddy System API (*get_free_pages* & friends); the buddy system allocator in turn uses pcp – per-cpu pagesets to perform allocation! >>. Without these per-CPU lists, memory allocation would scale poorly as the number of processors grows.

<<
Source: LKD3
…

Reasons for Using Per-CPU Data

There are several benefits to using per-CPU data. The first is the reduction in locking requirements. Depending on the semantics by which processors access the per-CPU data, you might not need any locking at all. Keep in mind that the “only this processor accesses this data” rule is only a programming convention. You need to ensure that the local processor accesses only its unique data. Nothing stops you from cheating.

Second, per-CPU data greatly reduces cache invalidation. This occurs as processors try to keep their caches in sync. If one processor manipulates data held in another processor’s cache, that processor must flush or otherwise update its cache. Constant cache invalidation is called thrashing the cache and wreaks havoc on system performance. The use of per-CPU data keeps cache effects to a minimum because processors ideally access only their own data. The percpu interface cache-aligns all data to ensure that accessing one processor’s data does not bring in another processor’s data on the same cache line << thus alleviating false-sharing issues as well >>.

Consequently, the use of per-CPU data often removes (or at least minimizes) the need for locking. The only safety requirement for the use of per-CPU data is disabling kernel preemption, which is much cheaper than locking, and the interface does so automatically.
Per-CPU data can safely be used from either interrupt or process context.
…

>>

Safe access to per-CPU data requires a couple of constraints, though: the thread working with the data cannot be preempted and it cannot be migrated while it manipulates per-CPU variables. If the thread is preempted, the thread that replaces it could try to work with the same variable; migration to another CPU could cause confusion for fairly obvious reasons.

To avoid these hazards, access to per-CPU variables is normally bracketed with calls to *get_cpu_var()* and *put_cpu_var()*; the *get_cpu_var()* call, along with providing the address for the processor's version of the variable, disables preemption.
<< Also: *Disabling kernel preemption, naturally disables migration as well.* >>

So code which obtains a reference to a per-CPU data will not be scheduled out of the CPU until it releases that reference. Needless to say, any such code must be atomic.

<<
As some "real" examples, kernel code (on ver 3.10.24) that calls *get_cpu_var()* :
Use cscope to see (formatted for readability):

```
Functions calling this function: get_cpu_var

  File                     Function                   Line
0 random.c                 get_random_int             1486 hash =
                                          get_cpu_var(get_random_int_hash);
1 netiucv.c                IUCV_DBF_TEXT_              117 char * __buf =
                                          get_cpu_var(iucv_dbf_txt_buf); \
2 fcoe.c                   fcoe_alloc_paged_crc_eof   1560 fps =
                                          &get_cpu_var(fcoe_percpu);
3 zsmalloc-main.c          zs_map_object              1007 area =
                                          &get_cpu_var(zs_map_area);
4 buffer.c                 invalidate_bh_lru          1393 struct bh_lru *b =
                                          &get_cpu_var(bh_lrus);
5 object.c                 fscache_enqueue_object      792
                                          &get_cpu_var(fscache_object_cong_wait);
6 inode.c                  get_next_ino                876 unsigned int *p =
                                          &get_cpu_var(last_ino);

--snip--

n netpoll.c                zap_completion_queue        294 struct softnet_data *sd
                                          = &get_cpu_var(softnet_data);
o ip_output.c              ip_send_unicast_reply      1511 inet =
                                          &get_cpu_var(unicast_sock);
p capability.c             audit_caps                   92 ent =
                                          &get_cpu_var(audit_cache);




Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function: get_cpu_var
Find this text string:
```

```
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:

>>
```

```
<<
```

File : [3.10.24] :include/linux/percpu.h

```
 ...
#define get_cpu_var(var) (*({                \
    preempt_disable();                \
    &__get_cpu_var(var); }))

/*
 * The weird & is necessary because sparse considers (void)(var) to be
 * a direct dereference of percpu variable (var).
 */
#define put_cpu_var(var) do {                \
    (void)&(var);                \
    preempt_enable();                \
} while (0)
...
>>
```

The conflict with realtime operation should be obvious: in the realtime world, anything that disables preemption is a possible source of unwanted latency. Realtime developers want the highest-priority process to run at all times; they have little patience for waiting while a low-priority thread gets around to releasing a per-CPU variable reference. In the past, this problem has been worked around by protecting per-CPU variables with spinlocks. These locks keep the code preemptable, but they wreck the scalability that per-CPU variables were created to provide and complicate the code. It has been clear for some time that a different solution would need to be found.

--snip--

The solution they came up with is surprisingly simple: whenever a process acquires a spinlock or obtains a CPU reference with *get_cpu()*, the scheduler will refrain from migrating that process to any other CPU. That process remains preemptable - code holding spinlocks can be preempted in the realtime world - but it will not be moved to another processor.

--snip--

Additional Resources

A brief introduction to per-cpu variables

SO :: How is percpu pointer implemented in kernel.

RCU

What is RCU, Fundamentally? [lwn]

From ELDD:

"...

The 2.6 kernel introduced another mechanism called Read-Copy Update (RCU), which yields improved performance when readers far outnumber writers. The basic idea is that reader threads can execute without locking.

Writer threads are more complex. They perform update operations on a copy of the data structure and replace the pointer that readers see. The original copy is maintained until the next context switch on all CPUs to ensure completion of all ongoing read operations.

Be aware that using RCU is more involved than using the primitives discussed thus far and should be used only if you are sure that it's the right tool for the job. RCU data structures and interface functions are defined in *include/linux/rcupdate.h*. There is ample documentation in *Documentation/RCU/**.

For an RCU usage example, look at *fs/dcache.c*. On Linux, each file is associated with directory entry information (stored in a structure called dentry), metadata information (stored in an inode), and actual data (stored in data blocks). Each time you operate on a file, the components in the file path are parsed, and the corresponding dentries are obtained. The dentries are kept cached in a data structure called the dcache, to speed up future operations. At any time, the number of dcache lookups is much more than dcache updates, so references to the dcache are protected using RCU primitives.

...”

From *Documentation/RCU/whatisRCU.txt*

Please note that the "What is RCU?" LWN series is an excellent place
to start learning about RCU:

1. What is RCU, Fundamentally? http://lwn.net/Articles/262464/
2. What is RCU? Part 2: Usage http://lwn.net/Articles/263130/
3. RCU part 3: the RCU API http://lwn.net/Articles/264090/
4. The RCU API, 2010 Edition http://lwn.net/Articles/418853/

…

Do see: https://www.kernel.org/doc/Documentation/RCU/rcu.txt

Unreliable Guide to Locking, Rusty Rusell (a bit old, but good).

Memory Barriers

Many processors and compilers reorder instructions to achieve optimal execution speeds. The reordering is done such that the new instruction stream is semantically equivalent to the original one.

However, if you are, for example, writing to memory mapped registers on an I/O device, instruction reordering can generate unexpected side effects. To prevent the processor from reordering instructions, you can insert a barrier in your code. The *wmb()* function inserts a road block that prevents writes from moving through it, *rmb()* provides a read barricade that disallows reads from crossing it, and *mb()* results in a read-write barrier.

<<

Memory barriers are also used as a (portable) way to defeat cache-coherency issues; inserting memory-barriers can ensure that pending memory writes have actually been written to main memory before proceeding.
(More below)..

>>

In addition to the CPU-to-hardware interactions referred to previously, memory barriers are also relevant for CPU-to-CPU interactions on SMP systems. If your CPU's data cache is operating in write-back mode (in which data is not copied from cache to memory until it's absolutely necessary), you might want to stall the instruction stream until the cache-to-memory queue is drained. This is relevant, for example, when you encounter instructions that acquire or release locks. Barriers are used in this scenario to obtain a consistent perception across CPUs.

We revisit memory barriers when we discuss PCI drivers in Chapter 10 and flash map drivers in Chapter 17. In the meanwhile, stop by *Documentation/memory-barriers.txt* for an explanation of different kinds of memory barriers.

<<

EXAMPLES of memory barrier usage

From Documentation/DMA-API-HOWTO.txt :

...

```
 IMPORTANT: Consistent DMA memory does not preclude the usage of
           proper memory barriers.  The CPU may reorder stores to
           consistent memory just as it may normal memory.  Example:
              if it is important for the device to see the first word
              of a descriptor updated before the second, you must do
              something like:

                   desc->word0 = address;
                   wmb();
                   desc->word1 = DESC_VALID;

           in order to get correct behavior on all platforms.
```

>>

<<

Source

…

All latest processors use an Out-Of-Order execution pipeline. What this simply means is that if current instruction has completed execution, it does not mean that the previous instruction has completed execution. As, the order in which the instructions are processed depends on the several factors like execution unit availability etc.. This is done for performance gain.

When implementing spin_locks() one needs to ensure this does not come in our way. We need to ensure that the previous lock holder is completely out of the critical section. So, we use these barrier instructions, which are again processor specific instructions which ensures that all the instructions before the barrier is completed.
…
>>

<<
An example of using a memory barrier across SMP shows up in the scheduling code:
kernel/sched.c

```
--snip--
/*
 * resched_task - mark a task 'to be rescheduled now'.
 *
 * On UP this means the setting of the need_resched flag, on SMP
   it might also involve a cross-CPU call to trigger the
   scheduler on the target CPU.
 */
#ifdef CONFIG_SMP

#ifndef tsk_is_polling
#define tsk_is_polling(t) test_tsk_thread_flag(t, TIF_POLLING_NRFLAG)
#endif

static void resched_task(struct task_struct *p)
{
    int cpu;

    assert_raw_spin_locked(&task_rq(p)->lock);

    if (test_tsk_need_resched(p))
        return;

    set_tsk_need_resched(p);

    cpu = task_cpu(p);
    if (cpu == smp_processor_id())
        return;

    /* NEED_RESCHED must be visible before we test polling */
    smp_mb();                     << the memory barrier >>
    if (!tsk_is_polling(p))
        smp_send_reschedule(cpu);
}
```

--snip--

>>

FYI:
In what situations might I need to insert memory barrier instructions? [ARM-specific]
Information on memory barriers from Wikipedia.

Debugging

Concurrency-related problems are generally hard to debug because they are usually difficult to reproduce. It's a good idea to enable SMP (CONFIG_SMP) and preemption (CONFIG_PREEMPT) while compiling and testing your code, even if your production kernel is going to run on a UP machine with preemption disabled.

There is a kernel configuration option under Kernel hacking called Spinlock and rw-lock debugging (CONFIG_DEBUG_SPINLOCK) that can help you catch some common spinlock errors. << And turn on CONFIG_DEBUG_MUTEXES to enable checks for mutex-related code. >>
Also available are tools such as lockmeter (http://oss.sgi.com/projects/lockmeter/) that collect lock-related statistics.

A common concurrency problem occurs when you forget to lock an access to a shared resource. This results in different threads "racing" through that access in an unregulated manner. The problem, called a race condition, might manifest in the form of occasional strange code behavior.

Another potential problem arises when you miss releasing held locks in certain code paths, resulting in deadlocks. To understand this, consider the following example:

```
spin_lock(&mylock);     /* Acquire lock */

/* ... Critical Section ... */

if (error) {            /* This error condition occurs rarely */
      return -EIO; /* Forgot to release the lock! */
}
```

```
spin_unlock(&mylock);    /* Release lock */
```

After the occurrence of the error condition, any thread trying to acquire mylock gets deadlocked, and the kernel might freeze.

If the problem first manifests months or years after you write the code, it'll be all the more tough to go back and debug it. To avoid such unpleasant encounters, concurrency logic should be designed when you architect your software.

<<

***Lockdep** – the Linux Runtime locking correctness validator.*

“Lockdep is very good at finding subtle locking problems which are difficult or impossible to expose with ordinary testing.”

To enable LOCKDEP, during kernel configuration,, select the option:
Kernel Hacking / Lock debugging / Lock debugging: prove locking correctness

From *lib/Kconfig.debug*

```
...
config PROVE_LOCKING
	bool "Lock debugging: prove locking correctness"
	depends on DEBUG_KERNEL && TRACE_IRQFLAGS_SUPPORT && STACKTRACE_SUPPORT 
&& LOCKDEP_SUPPORT
	select LOCKDEP
	select DEBUG_SPINLOCK
	select DEBUG_MUTEXES
	select DEBUG_LOCK_ALLOC
	select TRACE_IRQFLAGS
	default n
	help
	 This feature enables the kernel to prove that all locking
	 that occurs in the kernel runtime is mathematically
	 correct: that under no circumstance could an arbitrary (and
	 not yet triggered) combination of observed locking
	 sequences (on an arbitrary number of CPUs, running an
	 arbitrary number of tasks and interrupt contexts) cause a
	 deadlock.

	 In short, this feature enables the kernel to report locking
	 related deadlocks before they actually occur.

	 The proof does not depend on how hard and complex a
	 deadlock scenario would be to trigger: how many
	 participant CPUs, tasks and irq-contexts would be needed
	 for it to trigger. The proof also does not depend on
```

```
	  timing: if a race and a resulting deadlock is possible
	  theoretically (no matter how unlikely the race scenario
	  is), it will be proven so and will immediately be
	  reported by the kernel (once the event is observed that
	  makes the deadlock theoretically possible).

	  If a deadlock is impossible (i.e. the locking rules, as
	  observed by the kernel, are mathematically correct), the
	  kernel reports nothing.

	  NOTE: this feature can also be enabled for rwlocks, mutexes
	  and rwsems - in which case all dependencies between these
	  different locking variants are observed and mapped too, and
	  the proof of observed correctness is also maintained for an
	  arbitrary combination of these separate locking variants.

	  For more details, see Documentation/lockdep-design.txt.

config LOCKDEP
	bool
	depends on DEBUG_KERNEL && TRACE_IRQFLAGS_SUPPORT && STACKTRACE_SUPPORT
&& LOCKDEP_SUPPORT
	select STACKTRACE
	select FRAME_POINTER if !MIPS && !PPC && !ARM_UNWIND && !S390 && !
MICROBLAZE && !ARC
	select KALLSYMS
	select KALLSYMS_ALL
...
```

Additional Resources

SO :: How to use lockdep feature in linux kernel for deadlock detection

Also:

Userspace lockdep work exists *within* the kernel source tree:

tools/lib/lockdep

User-space lockdep

“...

The kernel's locking validator (often known as "lockdep") is one of the community's most useful pro-active debugging tools. Since its introduction in 2006, it has eliminated most deadlock-causing bugs from the system. Given that deadlocks can be extremely difficult to reproduce and diagnose, the result is a far more reliable kernel and happier users.

There is a shortage of equivalent tools for user-space programming, despite the fact that deadlock issues can happen there as well. As it happens, making lockdep available in user space may be far easier than almost anybody might have thought.
…"

SIDEBAR :: Using the kernel lockdep in userspace

You will require the kernel source tree

Steps:

```
cd <kernel-src-tree>/tools/lib/lockdep/
make

$ ls -F
Build                 include/              liblockdep.so@        lockdep.c
lockdep_states.h      preload.o             run_tests.sh*         common.c
liblockdep.a          liblockdep.so.5.0.0*  lockdep_internals.h   Makefile
rbtree.c              tests/                common.o              liblockdep-
in.o                  lockdep*              lockdep.o             preload.c
rbtree.o
$
$ cat lockdep
#!/bin/bash

LD_PRELOAD="./liblockdep.so $LD_PRELOAD" "$@"
$
```

Try it : on a simple but buggy matrix multiplication example which performs the incorrect AB-BA deadlock!

```
$ ./lockdep <...>/buggy_matrixmul
<...>//buggy_matrixmul: mutex process-shared is true.

====================================================
WARNING: possible circular locking dependency detected
liblockdep 5.0.0
----------------------------------------------------
buggy_matrixmul/13343 is trying to acquire lock:
0x201df98 (0x4040e0){....}, at: 0x4015fdS
```

```
but task is already holding lock:
0x201e008 (0x404160){....}, at: 0x4015f3S

which lock already depends on the new lock.


the existing dependency chain (in reverse order) is:

-> #1 (0x404160){....}:
./liblockdep.so(+0x2be1)[0x7fbc6d545be1]
./liblockdep.so(+0x4ed0)[0x7fbc6d547ed0]
./liblockdep.so(+0x50c9)[0x7fbc6d5480c9]
./liblockdep.so(+0x5893)[0x7fbc6d548893]
./liblockdep.so(+0x6884)[0x7fbc6d549884]
./liblockdep.so(lock_acquire+0x7d)[0x7fbc6d54aa40]
./liblockdep.so(pthread_mutex_lock+0x51)[0x7fbc6d54bc8e]
<...>//buggy_matrixmul[0x40153b]
/lib64/libpthread.so.0(+0x858e)[0x7fbc6d51158e]
/lib64/libc.so.6(clone+0x43)[0x7fbc6d4406a3]

-> #0 (0x4040e0){....}:
./liblockdep.so(+0x2be1)[0x7fbc6d545be1]
./liblockdep.so(+0x459a)[0x7fbc6d54759a]
./liblockdep.so(+0x4d38)[0x7fbc6d547d38]
./liblockdep.so(+0x50c9)[0x7fbc6d5480c9]
./liblockdep.so(+0x5893)[0x7fbc6d548893]
./liblockdep.so(+0x6884)[0x7fbc6d549884]
./liblockdep.so(lock_acquire+0x7d)[0x7fbc6d54aa40]
./liblockdep.so(pthread_mutex_lock+0x51)[0x7fbc6d54bc8e]
<...>/buggy_matrixmul[0x4015fd]
/lib64/libpthread.so.0(+0x858e)[0x7fbc6d51158e]
/lib64/libc.so.6(clone+0x43)[0x7fbc6d4406a3]

other info that might help us debug this:

 Possible unsafe locking scenario:

       CPU0                    CPU1
       ----                    ----
  lock(0x404160);
                               lock(0x4040e0);
                               lock(0x404160);
  lock(0x4040e0);

 *** DEADLOCK ***

1 lock held by buggy_matrixmul/13343:
 #0: 0x201e008 (0x404160){....}, at: 0x4015f3S

stack backtrace:
./liblockdep.so(+0x270e)[0x7fbc6d54570e]
./liblockdep.so(+0x4668)[0x7fbc6d547668]
./liblockdep.so(+0x4d38)[0x7fbc6d547d38]
./liblockdep.so(+0x50c9)[0x7fbc6d5480c9]
./liblockdep.so(+0x5893)[0x7fbc6d548893]
./liblockdep.so(+0x6884)[0x7fbc6d549884]
```

```
./liblockdep.so(lock_acquire+0x7d)[0x7fbc6d54aa40]
./liblockdep.so(pthread_mutex_lock+0x51)[0x7fbc6d54bc8e]
<...>/buggy_matrixmul[0x4015fd]
/lib64/libpthread.so.0(+0x858e)[0x7fbc6d51158e]
/lib64/libc.so.6(clone+0x43)[0x7fbc6d4406a3]

=====================================
WARNING: bad unlock balance detected!
liblockdep 5.0.0
-------------------------------------
buggy_matrixmul/13343 is trying to release lock (0x4040e0) at:
<...>/buggy_matrixmul() [0x401671]
but there are no more locks to release!

other info that might help us debug this:
1 lock held by buggy_matrixmul/13343:
 #0: 0x201e008 (0x404160){....}, at: 0x4015f3S

stack backtrace:
./liblockdep.so(+0x270e)[0x7fbc6d54570e]
./liblockdep.so(+0x6b06)[0x7fbc6d549b06]
./liblockdep.so(+0x7340)[0x7fbc6d54a340]
./liblockdep.so(lock_release+0x5d)[0x7fbc6d54aab2]
./liblockdep.so(pthread_mutex_unlock+0x39)[0x7fbc6d54bdb2]
<...>/buggy_matrixmul[0x401671]
/lib64/libpthread.so.0(+0x858e)[0x7fbc6d51158e]
/lib64/libc.so.6(clone+0x43)[0x7fbc6d4406a3]
0000000000222222222211111111113333333333
Sum =  400.000000
$
```

>>

Via "magic SysRq"

```
ARM # cat /proc/sys/kernel/sysrq
1
ARM # echo d > /proc/sysrq-trigger
sysrq: SysRq : Show Locks Held

Showing all locks held in the system:
3 locks held by sh/794:
 #0:  (sb_writers#4){.+.+}, at: [<802a307c>] vfs_write+0x164/0x178
 #1:  (rcu_read_lock){....}, at: [<804ec758>] __handle_sysrq+0x0/0x2a8
 #2:  (tasklist_lock){.+.+}, at: [<80177688>] debug_show_all_locks+0x40/0x1c0

=============================================

ARM #
```

Appendix A :: Internal Implementation of Spinlock on the ARMv6 and above

spin_lock() call graph (for Linux on ARM)

API / inline func *Source File*

```
spin_lock                             include/linux/spinlock.h
 raw_spin_lock                           -”-
   _raw_spin_lock                        -”-
    __raw_spin_lock                   kernel/spinlock.c
      __raw_spin_lock()               include/linux/spinlock_api_smp.h
        preempt_disable();               -”- << Note! K preemption disabled>>
        spin_acquire(...)                -”-
          do_raw_spin_lock            include/linux/spinlock.h
            __acquire(lock);             -”-
                      [ARM-specific ]
             arch_spin_lock(&lock->raw_lock); arch/arm/include/asm/spinlock.h
```

```
File : arch/arm/include/asm/spinlock.h
...
/*
 * ARMv6 ticket-based spin-locking.
 *
 * A memory barrier is required after we get a lock, and before we
 * release it, because V6 CPUs are assumed to have weakly ordered
 * memory.
 */

--snip--

static inline void arch_spin_lock(arch_spinlock_t *lock)
{
    unsigned long tmp;
    u32 newval;
    arch_spinlock_t lockval;

    __asm__ __volatile__(
"1: ldrex   %0, [%3]\n"
"   add %1, %0, %4\n"
"   strex   %2, %1, [%3]\n"
"   teq %2, #0\n"
"   bne 1b"
    : "=&r" (lockval), "=&r" (newval), "=&r" (tmp)
    : "r" (&lock->slock), "I" (1 << TICKET_SHIFT)
    : "cc");

    while (lockval.tickets.next != lockval.tickets.owner) {
        wfe();
        lockval.tickets.owner = ACCESS_ONCE(lock->tickets.owner);
    }

    smp_mb();
}
...
```

Source : spin_lock implementation in ARM linux

…
spin_locks are usually implemented with some support from hardware. It is not a purely software concept in the implementation. It is usually a assembly code. ARM processor provides two special instructions which are primarily for implementation of spin_locks. These instructions are LDREX and STREX. The older versions of the ARM processor had the SWP instruction to implement a spin_lock. The x86 architecture also provides the cmpxchg instruction which is used for this.

Before understanding the implementation details, let's first understand a few basic instructions that are used in the implementation.

IMPORTANT CONCEPTS THAT ARE NEEDED TO UNDERSTAND SPIN_LOCK BETTER:

wfe: Wait For Event.
This is an ARM instruction which puts the ARM core into a lower power state. The core logic is off but the wake-up mechanism and the RAM arrays are kept on. This is usally the first thing that the cpu does when it goes into idle. The wake-up from this (is) very very fast, hence it is used in spinlocks.

The wake is from a external event from another processor or interrupt. If you notice carefully we are technically not spinning during a spinlock ;-). Spinning is such a waste of power. But, we are not getting scheduled out as well. So, from the process point of view it is still holding the CPU.

sev: Send Event. This to send an event to another Processor in the system to wake it up from WFE state to start executing code.

Barriers: All latest processors use an Out-Of-Order execution pipeline. What this simply means is that if current instruction has completed execution, it does not mean that the previous instruction has completed execution. As, the order in which the instructions are processed depends on the several factors like execution unit availability etc.. This is done for performance gain. When implementing spin_locks() one needs to ensure this does not come in our way. We need to ensure that the previous lock holder is completely out of the critical section. So, we use these barrier instructions, which are again processor specific instructions which ensures that all the instructions before the barrier is completed.

ldrex, strex:

These instructions help in automatically updated the memory from 0 to 1. The older instructions like SWP was a single instruction with tries to change the memory in one go automatically In the newer ARM architectures this is split into two instructions and some logic can be applied in between these instructions. These instructions have been explained in detail in the ARM manuals.

Now, lets have a overview of the implementation itself.
In Linux, the ARM implementation can be seen *arch/arm/include/asm/spinlock.h* .

Once the above concepts are understood the code below becomes very much self-explanatory. The code for implementing this keeps constantly improving, but the fundamental concept behind the implementation does not change drastically.

```
1:  ldrex     <<< load the value exclusively >>>
    teq       <<< test if equal to zero >>>
    wfe <<< wait for event if not zero. which means someone is holding it >>>
    strexeq   <<< no one is holding the lock try to store 1. >>>
    teq       <<< check if stored correctly >>>
    bne 1b    <<< if not stored correctly go and and try again >>>
```

Appendix B :: Internal Implementation of Spinlock on x86

***FAQ:* How are spinlocks, mutex locks, semaphores, monitors, etc *actually implemented*?**

A. It can't be done directly by C/C++ code; it requires special machine instructions (within the underlying processor ISA (Instruction Set Architecture)) to achieve true atomicity. All processor's ISA will provide at least one machine-level instruction to do so. Typically, a "test-and-set" and/or a "compare-exchange" (or 'compare-swap') type of machine instruction.

Please see:

- The section *"Scenarios 2: Atomic compare-and-swap (CAS)" in this article* (PPC-biased) to see more details
- *"Peeking Under the Hood"* (x86-biased)

***SIDEBAR :: Intel* CMPCHXG**

Source: Intel® 64 and IA-32 Architectures
Software Developer's Manual
Volume 2A:
Instruction Set Reference, A-M . *Download link from Intel*.

The CMPXCHG machine instruction provides the required "test-and-set" atomic operation on Intel processors.

Source

CMPXCHG: Compare and Exchange

```
CMPXCHG r/m8,reg8             ; 0F B0 /r    *        [PENT]
CMPXCHG r/m16,reg16           ; o16 0F B1 /r         [PENT]
CMPXCHG r/m32,reg32           ; o32 0F B1 /r         [PENT]
```

```
* Opcode is 0F B0
```

```
/r – Indicates that the ModR/M byte of the instruction contains a register
operand and an r/m (register/memory) operand.
```

CMPXCHG compares its destination (first) operand to the value in AL, AX or EAX (depending on the size of the instruction). If they are equal, it copies its source (second) operand into the destination and sets the zero flag. Otherwise, it clears the zero flag and leaves the destination alone.

<< From Intel's manual:

(Showing the 32-bit ver as an example):

```
CMPXCHG r/m32, r32          Compare EAX with r/m32. If equal, ZF
                     is set and r32 is loaded into r/m32. Else,
                     clear ZF and load r/m32 into EAX.
...
```

Operation

```
(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word,
doubleword, or quadword comparison is being performed *)

IF accumulator = DEST
  THEN
     ZF ← 1;
     DEST ← SRC;
  ELSE
     ZF ← 0;
     accumulator ← DEST;
FI;
...
```

Also, multibyte (8 and 16 byte) compare-exchange can be performed with the CMPXCHG8B and CMPXCHG16B instructions.

>>

CMPXCHG is intended to be used for atomic operations in multitasking or multiprocessor environments. To safely update a value in shared memory, for example, you might load the value into EAX, load the updated value into EBX, and then execute the instruction `lock cmpxchg [value],ebx`.

If value has not changed since being loaded, it is updated with your desired new value, and the zero flag is set to let you know it has worked. (The LOCK prefix prevents another processor doing anything in the middle of this operation: it guarantees atomicity.)

However, if another processor has modified the value in between your load and your attempted store, the store does not happen, and you are notified of the failure by a cleared zero flag, so you can go round and try again.

Also: a brief interesting explanation on the implementation of spinlocks on ARM processors. If interested, please see *"Appendix A :: Internal Implementation of Spinlock on the ARMv6 and above"* at the end of this module.

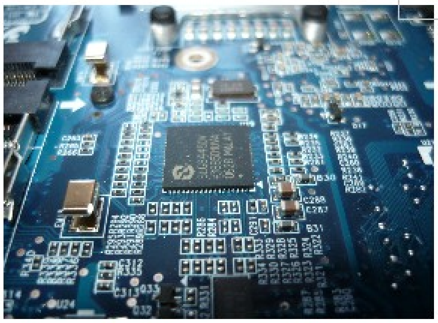
Recent [3.15 kernel]

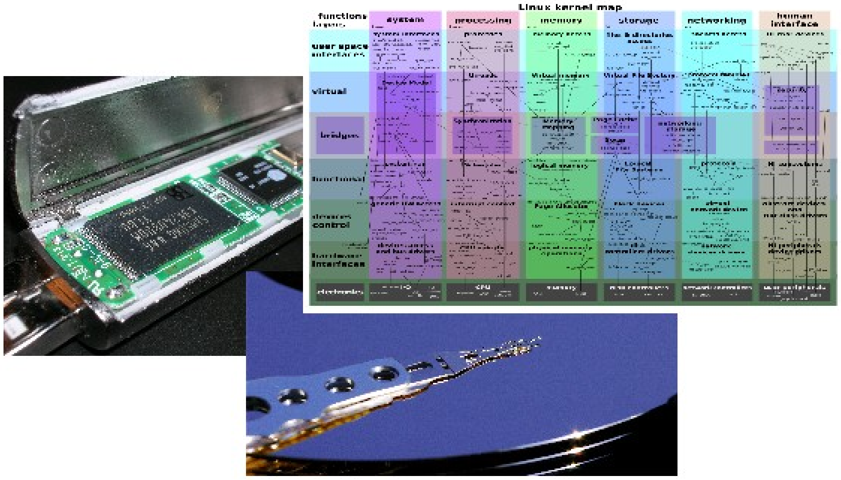
Spinlocks have a new implementation – the so-called "MCS locks" and "qspinlocks. Slated for release in 3.15 Linux kernel.

Please see this LWN article for details:
"MCS locks and qspinlocks", Jon Corbet, May 2014.

Update
Indeed, on http://kernelnewbies.org/Linux_3.15 :

… Introduce cancelable MCS lock, it is a simple spinlock with the desirable properties of being fair, and with each CPU trying to acquire the lock spinning on a local variable. It avoids expensive cache bouncings that common test-and-set spinlock implementations incur commit ...




Linux kernel map