```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
	"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="LKLockingGuide">
 <bookinfo>
  <title>Unreliable Guide To Locking</title>

  <authorgroup>
   <author>
    <firstname>Rusty</firstname>
    <surname>Russell</surname>
    <affiliation>
     <address>
      <email>rusty@rustcorp.com.au</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>

  <copyright>
   <year>2003</year>
   <holder>Rusty Russell</holder>
  </copyright>

  <legalnotice>
   <para>
     This documentation is free software; you can redistribute
     it and/or modify it under the terms of the GNU General Public
     License as published by the Free Software Foundation; either
     version 2 of the License, or (at your option) any later
     version.
   </para>

   <para>
     This program is distributed in the hope that it will be
     useful, but WITHOUT ANY WARRANTY; without even the implied
     warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
     See the GNU General Public License for more details.
   </para>

   <para>
     You should have received a copy of the GNU General Public
     License along with this program; if not, write to the Free
     Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
     MA 02111-1307 USA
   </para>

   <para>
     For more details see the file COPYING in the source
     distribution of Linux.
   </para>
  </legalnotice>
 </bookinfo>

 <toc></toc>
```

```
<chapter id="intro">
 <title>Introduction</title>
 <para>
   Welcome, to Rusty's Remarkably Unreliable Guide to Kernel
   Locking issues.  This document describes the locking systems in
   the Linux Kernel in 2.6.
 </para>
 <para>
   With the wide availability of HyperThreading, and <firstterm
   linkend="gloss-preemption">preemption </firstterm> in the Linux
   Kernel, everyone hacking on the kernel needs to know the
   fundamentals of concurrency and locking for
   <firstterm linkend="gloss-smp"><acronym>SMP</acronym></firstterm>.
 </para>
</chapter>

 <chapter id="races">
  <title>The Problem With Concurrency</title>
  <para>
    (Skip this if you know what a Race Condition is).
  </para>
  <para>
    In a normal program, you can increment a counter like so:
  </para>
  <programlisting>
    very_important_count++;
  </programlisting>

  <para>
    This is what they would expect to happen:
  </para>

  <table>
   <title>Expected Results</title>

   <tgroup cols="2" align="left">

    <thead>
     <row>
      <entry>Instance 1</entry>
      <entry>Instance 2</entry>
     </row>
    </thead>

    <tbody>
     <row>
      <entry>read very_important_count (5)</entry>
      <entry></entry>
     </row>
     <row>
      <entry>add 1 (6)</entry>
      <entry></entry>
     </row>
     <row>
      <entry>write very_important_count (6)</entry>
      <entry></entry>
```

```
   </row>
   <row>
    <entry></entry>
    <entry>read very_important_count (6)</entry>
   </row>
   <row>
    <entry></entry>
    <entry>add 1 (7)</entry>
   </row>
   <row>
    <entry></entry>
    <entry>write very_important_count (7)</entry>
   </row>
  </tbody>

 </tgroup>
</table>

<para>
 This is what might happen:
</para>

<table>
 <title>Possible Results</title>

 <tgroup cols="2" align="left">
  <thead>
   <row>
    <entry>Instance 1</entry>
    <entry>Instance 2</entry>
   </row>
  </thead>

  <tbody>
   <row>
    <entry>read very_important_count (5)</entry>
    <entry></entry>
   </row>
   <row>
    <entry></entry>
    <entry>read very_important_count (5)</entry>
   </row>
   <row>
    <entry>add 1 (6)</entry>
    <entry></entry>
   </row>
   <row>
    <entry></entry>
    <entry>add 1 (6)</entry>
   </row>
   <row>
    <entry>write very_important_count (6)</entry>
    <entry></entry>
   </row>
   <row>
    <entry></entry>
```

```
        <entry>write very_important_count (6)</entry>
       </row>
      </tbody>
     </tgroup>
    </table>

    <sect1 id="race-condition">
    <title>Race Conditions and Critical Regions</title>
    <para>
      This overlap, where the result depends on the
      relative timing of multiple tasks, is called a <firstterm>race
condition</firstterm>.
      The piece of code containing the concurrency issue is called a
      <firstterm>critical region</firstterm>.  And especially since Linux
starting running
      on SMP machines, they became one of the major issues in kernel
      design and implementation.
    </para>
    <para>
      Preemption can have the same effect, even if there is only one
      CPU: by preempting one task during the critical region, we have
      exactly the same race condition.  In this case the thread which
      preempts might run the critical region itself.
    </para>
    <para>
      The solution is to recognize when these simultaneous accesses
      occur, and use locks to make sure that only one instance can
      enter the critical region at any time.  There are many
      friendly primitives in the Linux kernel to help you do this.
      And then there are the unfriendly primitives, but I'll pretend
      they don't exist.
    </para>
    </sect1>
  </chapter>

  <chapter id="locks">
   <title>Locking in the Linux Kernel</title>

   <para>
     If I could give you one piece of advice: never sleep with anyone
     crazier than yourself.  But if I had to give you advice on
     locking: <emphasis>keep it simple</emphasis>.
   </para>

   <para>
     Be reluctant to introduce new locks.
   </para>

   <para>
     Strangely enough, this last one is the exact reverse of my advice when
     you <emphasis>have</emphasis> slept with someone crazier than yourself.
     And you should think about getting a big dog.
   </para>

   <sect1 id="lock-intro">
   <title>Two Main Types of Kernel Locks: Spinlocks and Mutexes</title>
```

```
   <para>
     There are two main types of kernel locks.  The fundamental type
     is the spinlock
     (<filename class="headerfile">include/asm/spinlock.h</filename>),
     which is a very simple single-holder lock: if you can't get the
     spinlock, you keep trying (spinning) until you can.  Spinlocks are
     very small and fast, and can be used anywhere.
   </para>
   <para>
     The second type is a mutex
     (<filename class="headerfile">include/linux/mutex.h</filename>): it
     is like a spinlock, but you may block holding a mutex.
     If you can't lock a mutex, your task will suspend itself, and be woken
     up when the mutex is released.  This means the CPU can do something
     else while you are waiting.  There are many cases when you simply
     can't sleep (see <xref linkend="sleeping-things"/>), and so have to
     use a spinlock instead.
   </para>
   <para>
     Neither type of lock is recursive: see
     <xref linkend="deadlock"/>.
   </para>
   </sect1>

   <sect1 id="uniprocessor">
    <title>Locks and Uniprocessor Kernels</title>

    <para>
      For kernels compiled without <symbol>CONFIG_SMP</symbol>, and
      without <symbol>CONFIG_PREEMPT</symbol> spinlocks do not exist at
      all.  This is an excellent design decision: when no-one else can
      run at the same time, there is no reason to have a lock.
    </para>

    <para>
      If the kernel is compiled without <symbol>CONFIG_SMP</symbol>,
      but <symbol>CONFIG_PREEMPT</symbol> is set, then spinlocks
      simply disable preemption, which is sufficient to prevent any
      races.  For most purposes, we can think of preemption as
      equivalent to SMP, and not worry about it separately.
    </para>

    <para>
      You should always test your locking code with <symbol>CONFIG_SMP</symbol>
      and <symbol>CONFIG_PREEMPT</symbol> enabled, even if you don't have an SMP
test box, because it
      will still catch some kinds of locking bugs.
    </para>

    <para>
      Mutexes still exist, because they are required for
      synchronization between <firstterm linkend="gloss-usercontext">user
      contexts</firstterm>, as we will see below.
    </para>
   </sect1>
```

```
  <sect1 id="usercontextlocking">
   <title>Locking Only In User Context</title>

   <para>
     If you have a data structure which is only ever accessed from
     user context, then you can use a simple mutex
     (<filename>include/linux/mutex.h</filename>) to protect it.  This
     is the most trivial case: you initialize the mutex.  Then you can
     call <function>mutex_lock_interruptible()</function> to grab the mutex,
     and <function>mutex_unlock()</function> to release it.  There is also a
     <function>mutex_lock()</function>, which should be avoided, because it
     will not return if a signal is received.
   </para>

   <para>
     Example: <filename>net/netfilter/nf_sockopt.c</filename> allows
     registration of new <function>setsockopt()</function> and
     <function>getsockopt()</function> calls, with
     <function>nf_register_sockopt()</function>.  Registration and
     de-registration are only done on module load and unload (and boot
     time, where there is no concurrency), and the list of registrations
     is only consulted for an unknown <function>setsockopt()</function>
     or <function>getsockopt()</function> system call.  The
     <varname>nf_sockopt_mutex</varname> is perfect to protect this,
     especially since the setsockopt and getsockopt calls may well
     sleep.
   </para>
 </sect1>

 <sect1 id="lock-user-bh">
  <title>Locking Between User Context and Softirqs</title>

  <para>
    If a <firstterm linkend="gloss-softirq">softirq</firstterm> shares
    data with user context, you have two problems.  Firstly, the current
    user context can be interrupted by a softirq, and secondly, the
    critical region could be entered from another CPU.  This is where
    <function>spin_lock_bh()</function>
    (<filename class="headerfile">include/linux/spinlock.h</filename>) is
    used.  It disables softirqs on that CPU, then grabs the lock.
    <function>spin_unlock_bh()</function> does the reverse.  (The
    '_bh' suffix is a historical reference to "Bottom Halves", the
    old name for software interrupts.  It should really be
    called spin_lock_softirq()' in a perfect world).
  </para>

  <para>
    Note that you can also use <function>spin_lock_irq()</function>
    or <function>spin_lock_irqsave()</function> here, which stop
    hardware interrupts as well: see <xref linkend="hardirq-context"/>.
  </para>

  <para>
    This works perfectly for <firstterm linkend="gloss-up"><acronym>UP
    </acronym></firstterm> as well: the spin lock vanishes, and this macro
```

```
    simply becomes <function>local_bh_disable()</function>
    (<filename class="headerfile">include/linux/interrupt.h</filename>), which
    protects you from the softirq being run.
  </para>
 </sect1>

 <sect1 id="lock-user-tasklet">
  <title>Locking Between User Context and Tasklets</title>

  <para>
    This is exactly the same as above, because <firstterm
    linkend="gloss-tasklet">tasklets</firstterm> are actually run
    from a softirq.
  </para>
 </sect1>

 <sect1 id="lock-user-timers">
  <title>Locking Between User Context and Timers</title>

  <para>
    This, too, is exactly the same as above, because <firstterm
    linkend="gloss-timers">timers</firstterm> are actually run from
    a softirq.  From a locking point of view, tasklets and timers
    are identical.
  </para>
 </sect1>

 <sect1 id="lock-tasklets">
  <title>Locking Between Tasklets/Timers</title>

  <para>
    Sometimes a tasklet or timer might want to share data with
    another tasklet or timer.
  </para>

  <sect2 id="lock-tasklets-same">
   <title>The Same Tasklet/Timer</title>
   <para>
     Since a tasklet is never run on two CPUs at once, you don't
     need to worry about your tasklet being reentrant (running
     twice at once), even on SMP.
   </para>
  </sect2>

  <sect2 id="lock-tasklets-different">
   <title>Different Tasklets/Timers</title>
   <para>
     If another tasklet/timer wants
     to share data with your tasklet or timer , you will both need to use
     <function>spin_lock()</function> and
     <function>spin_unlock()</function> calls.
     <function>spin_lock_bh()</function> is
     unnecessary here, as you are already in a tasklet, and
     none will be run on the same CPU.
   </para>
  </sect2>
```

```
  </sect1>

  <sect1 id="lock-softirqs">
   <title>Locking Between Softirqs</title>

   <para>
     Often a softirq might
     want to share data with itself or a tasklet/timer.
   </para>

   <sect2 id="lock-softirqs-same">
    <title>The Same Softirq</title>

    <para>
      The same softirq can run on the other CPUs: you can use a
      per-CPU array (see <xref linkend="per-cpu"/>) for better
      performance.  If you're going so far as to use a softirq,
      you probably care about scalable performance enough
      to justify the extra complexity.
    </para>

    <para>
      You'll need to use <function>spin_lock()</function> and
      <function>spin_unlock()</function> for shared data.
    </para>
   </sect2>

   <sect2 id="lock-softirqs-different">
    <title>Different Softirqs</title>

    <para>
      You'll need to use <function>spin_lock()</function> and
      <function>spin_unlock()</function> for shared data, whether it
      be a timer, tasklet, different softirq or the same or another
      softirq: any of them could be running on a different CPU.
    </para>
   </sect2>
  </sect1>
 </chapter>

 <chapter id="hardirq-context">
  <title>Hard IRQ Context</title>

  <para>
    Hardware interrupts usually communicate with a
    tasklet or softirq.  Frequently this involves putting work in a
    queue, which the softirq will take out.
  </para>

  <sect1 id="hardirq-softirq">
   <title>Locking Between Hard IRQ and Softirqs/Tasklets</title>

   <para>
     If a hardware irq handler shares data with a softirq, you have
     two concerns.  Firstly, the softirq processing can be
     interrupted by a hardware interrupt, and secondly, the
```

```
    critical region could be entered by a hardware interrupt on
    another CPU.  This is where <function>spin_lock_irq()</function> is
    used.  It is defined to disable interrupts on that cpu, then grab
    the lock. <function>spin_unlock_irq()</function> does the reverse.
  </para>

  <para>
    The irq handler does not to use
    <function>spin_lock_irq()</function>, because the softirq cannot
    run while the irq handler is running: it can use
    <function>spin_lock()</function>, which is slightly faster.  The
    only exception would be if a different hardware irq handler uses
    the same lock: <function>spin_lock_irq()</function> will stop
    that from interrupting us.
  </para>

  <para>
    This works perfectly for UP as well: the spin lock vanishes,
    and this macro simply becomes <function>local_irq_disable()</function>
    (<filename class="headerfile">include/asm/smp.h</filename>), which
    protects you from the softirq/tasklet/BH being run.
  </para>

  <para>
    <function>spin_lock_irqsave()</function>
    (<filename>include/linux/spinlock.h</filename>) is a variant
    which saves whether interrupts were on or off in a flags word,
    which is passed to <function>spin_unlock_irqrestore()</function>.  This
    means that the same code can be used inside an hard irq handler (where
    interrupts are already off) and in softirqs (where the irq
    disabling is required).
  </para>

  <para>
    Note that softirqs (and hence tasklets and timers) are run on
    return from hardware interrupts, so
    <function>spin_lock_irq()</function> also stops these.  In that
    sense, <function>spin_lock_irqsave()</function> is the most
    general and powerful locking function.
  </para>

 </sect1>
 <sect1 id="hardirq-hardirq">
  <title>Locking Between Two Hard IRQ Handlers</title>
  <para>
    It is rare to have to share data between two IRQ handlers, but
    if you do, <function>spin_lock_irqsave()</function> should be
    used: it is architecture-specific whether all interrupts are
    disabled inside irq handlers themselves.
  </para>
 </sect1>

</chapter>

<chapter id="cheatsheet">
 <title>Cheat Sheet For Locking</title>
```

```
   <para>
     Pete Zaitcev gives the following summary:
   </para>
   <itemizedlist>
      <listitem>
        <para>
          If you are in a process context (any syscall) and want to
        lock other process out, use a mutex.  You can take a mutex
        and sleep (<function>copy_from_user*(</function> or
        <function>kmalloc(x,GFP_KERNEL)</function>).
      </para>
      </listitem>
      <listitem>
        <para>
        Otherwise (== data can be touched in an interrupt), use
        <function>spin_lock_irqsave()</function> and
        <function>spin_unlock_irqrestore()</function>.
        </para>
      </listitem>
      <listitem>
        <para>
        Avoid holding spinlock for more than 5 lines of code and
        across any function call (except accessors like
        <function>readb</function>).
        </para>
      </listitem>
    </itemizedlist>

   <sect1 id="minimum-lock-reqirements">
   <title>Table of Minimum Requirements</title>

   <para> The following table lists the <emphasis>minimum</emphasis>
        locking requirements between various contexts.  In some cases,
        the same context can only be running on one CPU at a time, so
        no locking is required for that context (eg. a particular
        thread can only run on one CPU at a time, but if it needs
        shares data with another thread, locking is required).
   </para>
   <para>
        Remember the advice above: you can always use
        <function>spin_lock_irqsave()</function>, which is a superset
        of all other spinlock primitives.
   </para>

   <table>
<title>Table of Locking Requirements</title>
<tgroup cols="11">
<tbody>

<row>
<entry></entry>
<entry>IRQ Handler A</entry>
<entry>IRQ Handler B</entry>
<entry>Softirq A</entry>
<entry>Softirq B</entry>
<entry>Tasklet A</entry>
```

```
<entry>Tasklet B</entry>
<entry>Timer A</entry>
<entry>Timer B</entry>
<entry>User Context A</entry>
<entry>User Context B</entry>
</row>

<row>
<entry>IRQ Handler A</entry>
<entry>None</entry>
</row>

<row>
<entry>IRQ Handler B</entry>
<entry>SLIS</entry>
<entry>None</entry>
</row>

<row>
<entry>Softirq A</entry>
<entry>SLI</entry>
<entry>SLI</entry>
<entry>SL</entry>
</row>

<row>
<entry>Softirq B</entry>
<entry>SLI</entry>
<entry>SLI</entry>
<entry>SL</entry>
<entry>SL</entry>
</row>

<row>
<entry>Tasklet A</entry>
<entry>SLI</entry>
<entry>SLI</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>None</entry>
</row>

<row>
<entry>Tasklet B</entry>
<entry>SLI</entry>
<entry>SLI</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>None</entry>
</row>

<row>
<entry>Timer A</entry>
<entry>SLI</entry>
<entry>SLI</entry>
```

```
<entry>SL</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>None</entry>
</row>

<row>
<entry>Timer B</entry>
<entry>SLI</entry>
<entry>SLI</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>SL</entry>
<entry>None</entry>
</row>

<row>
<entry>User Context A</entry>
<entry>SLI</entry>
<entry>SLI</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>None</entry>
</row>

<row>
<entry>User Context B</entry>
<entry>SLI</entry>
<entry>SLI</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>SLBH</entry>
<entry>MLI</entry>
<entry>None</entry>
</row>

</tbody>
</tgroup>
</table>

   <table>
<title>Legend for Locking Requirements Table</title>
<tgroup cols="2">
<tbody>

<row>
```

```
<entry>SLIS</entry>
<entry>spin_lock_irqsave</entry>
</row>
<row>
<entry>SLI</entry>
<entry>spin_lock_irq</entry>
</row>
<row>
<entry>SL</entry>
<entry>spin_lock</entry>
</row>
<row>
<entry>SLBH</entry>
<entry>spin_lock_bh</entry>
</row>
<row>
<entry>MLI</entry>
<entry>mutex_lock_interruptible</entry>
</row>

</tbody>
</tgroup>
</table>

</sect1>
</chapter>

<chapter id="trylock-functions">
 <title>The trylock Functions</title>
  <para>
   There are functions that try to acquire a lock only once and immediately
   return a value telling about success or failure to acquire the lock.
   They can be used if you need no access to the data protected with the lock
   when some other thread is holding the lock. You should acquire the lock
   later if you then need access to the data protected with the lock.
  </para>

  <para>
    <function>spin_trylock()</function> does not spin but returns non-zero if
    it acquires the spinlock on the first try or 0 if not. This function can
    be used in all contexts like <function>spin_lock</function>: you must have
    disabled the contexts that might interrupt you and acquire the spin lock.
  </para>

  <para>
    <function>mutex_trylock()</function> does not suspend your task
    but returns non-zero if it could lock the mutex on the first try
    or 0 if not. This function cannot be safely used in hardware or software
    interrupt contexts despite not sleeping.
  </para>
</chapter>

  <chapter id="Examples">
   <title>Common Examples</title>
    <para>
Let's step through a simple example: a cache of number to name
```

mappings. The cache keeps a count of how often each of the objects is
used, and when it gets full, throws out the least used one.

    </para>

   <sect1 id="examples-usercontext">
    <title>All In User Context</title>
    <para>
For our first example, we assume that all operations are in user
context (ie. from system calls), so we can sleep. This means we can
use a mutex to protect the cache and all the objects within
it. Here's the code:
    </para>

    <programlisting>
```
#include &lt;linux/list.h&gt;
#include &lt;linux/slab.h&gt;
#include &lt;linux/string.h&gt;
#include &lt;linux/mutex.h&gt;
#include &lt;asm/errno.h&gt;

struct object
{
        struct list_head list;
        int id;
        char name[32];
        int popularity;
};

/* Protects the cache, cache_num, and the objects within it */
static DEFINE_MUTEX(cache_lock);
static LIST_HEAD(cache);
static unsigned int cache_num = 0;
#define MAX_CACHE_SIZE 10

/* Must be holding cache_lock */
static struct object *__cache_find(int id)
{
        struct object *i;

        list_for_each_entry(i, &amp;cache, list)
                if (i-&gt;id == id) {
                        i-&gt;popularity++;
                        return i;
                }
        return NULL;
}

/* Must be holding cache_lock */
static void __cache_delete(struct object *obj)
{
        BUG_ON(!obj);
        list_del(&amp;obj-&gt;list);
        kfree(obj);
        cache_num--;
}
```

```
/* Must be holding cache_lock */
static void __cache_add(struct object *obj)
{
        list_add(&amp;obj-&gt;list, &amp;cache);
        if (++cache_num > MAX_CACHE_SIZE) {
                struct object *i, *outcast = NULL;
                list_for_each_entry(i, &amp;cache, list) {
                        if (!outcast || i-&gt;popularity &lt;
outcast-&gt;popularity)
                                outcast = i;
                }
                __cache_delete(outcast);
        }
}

int cache_add(int id, const char *name)
{
        struct object *obj;

        if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
                return -ENOMEM;

        strlcpy(obj-&gt;name, name, sizeof(obj-&gt;name));
        obj-&gt;id = id;
        obj-&gt;popularity = 0;

        mutex_lock(&amp;cache_lock);
        __cache_add(obj);
        mutex_unlock(&amp;cache_lock);
        return 0;
}

void cache_delete(int id)
{
        mutex_lock(&amp;cache_lock);
        __cache_delete(__cache_find(id));
        mutex_unlock(&amp;cache_lock);
}

int cache_find(int id, char *name)
{
        struct object *obj;
        int ret = -ENOENT;

        mutex_lock(&amp;cache_lock);
        obj = __cache_find(id);
        if (obj) {
                ret = 0;
                strcpy(name, obj-&gt;name);
        }
        mutex_unlock(&amp;cache_lock);
        return ret;
}
</programlisting>
```

```
    <para>
Note that we always make sure we have the cache_lock when we add,
delete, or look up the cache: both the cache infrastructure itself and
the contents of the objects are protected by the lock.  In this case
it's easy, since we copy the data for the user, and never let them
access the objects directly.
    </para>
    <para>
There is a slight (and common) optimization here: in
<function>cache_add</function> we set up the fields of the object
before grabbing the lock.  This is safe, as no-one else can access it
until we put it in cache.
    </para>
    </sect1>

   <sect1 id="examples-interrupt">
    <title>Accessing From Interrupt Context</title>
    <para>
Now consider the case where <function>cache_find</function> can be
called from interrupt context: either a hardware interrupt or a
softirq.  An example would be a timer which deletes object from the
cache.
    </para>
    <para>
The change is shown below, in standard patch format: the
<symbol>-</symbol> are lines which are taken away, and the
<symbol>+</symbol> are lines which are added.
    </para>
<programlisting>
--- cache.c.usercontext 2003-12-09 13:58:54.000000000 +1100
+++ cache.c.interrupt   2003-12-09 14:07:49.000000000 +1100
@@ -12,7 +12,7 @@
         int popularity;
 };

-static DEFINE_MUTEX(cache_lock);
+static DEFINE_SPINLOCK(cache_lock);
 static LIST_HEAD(cache);
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10
@@ -55,6 +55,7 @@
 int cache_add(int id, const char *name)
 {
         struct object *obj;
+        unsigned long flags;

         if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
                 return -ENOMEM;
@@ -63,30 +64,33 @@
         obj-&gt;id = id;
         obj-&gt;popularity = 0;

-        mutex_lock(&amp;cache_lock);
+        spin_lock_irqsave(&amp;cache_lock, flags);
         __cache_add(obj);
-        mutex_unlock(&amp;cache_lock);
```

```
+        spin_unlock_irqrestore(&amp;cache_lock, flags);
         return 0;
 }

 void cache_delete(int id)
 {
-        mutex_lock(&amp;cache_lock);
+        unsigned long flags;
+
+        spin_lock_irqsave(&amp;cache_lock, flags);
         __cache_delete(__cache_find(id));
-        mutex_unlock(&amp;cache_lock);
+        spin_unlock_irqrestore(&amp;cache_lock, flags);
 }

 int cache_find(int id, char *name)
 {
         struct object *obj;
         int ret = -ENOENT;
+        unsigned long flags;

-        mutex_lock(&amp;cache_lock);
+        spin_lock_irqsave(&amp;cache_lock, flags);
         obj = __cache_find(id);
         if (obj) {
                 ret = 0;
                 strcpy(name, obj-&gt;name);
         }
-        mutex_unlock(&amp;cache_lock);
+        spin_unlock_irqrestore(&amp;cache_lock, flags);
         return ret;
 }
</programlisting>

    <para>
Note that the <function>spin_lock_irqsave</function> will turn off
interrupts if they are on, otherwise does nothing (if we are already
in an interrupt handler), hence these functions are safe to call from
any context.
    </para>
    <para>
Unfortunately, <function>cache_add</function> calls
<function>kmalloc</function> with the <symbol>GFP_KERNEL</symbol>
flag, which is only legal in user context.  I have assumed that
<function>cache_add</function> is still only called in user context,
otherwise this should become a parameter to
<function>cache_add</function>.
    </para>
  </sect1>
   <sect1 id="examples-refcnt">
    <title>Exposing Objects Outside This File</title>
    <para>
If our objects contained more information, it might not be sufficient
to copy the information in and out: other parts of the code might want
to keep pointers to these objects, for example, rather than looking up
the id every time.  This produces two problems.
```

    </para>
    <para>
The first problem is that we use the <symbol>cache_lock</symbol> to protect objects: we'd need to make this non-static so the rest of the code can use it. This makes locking trickier, as it is no longer all in one place.
    </para>
    <para>
The second problem is the lifetime problem: if another structure keeps a pointer to an object, it presumably expects that pointer to remain valid. Unfortunately, this is only guaranteed while you hold the lock, otherwise someone might call <function>cache_delete</function> and even worse, add another object, re-using the same address.
    </para>
    <para>
As there is only one lock, you can't hold it forever: no-one else would get any work done.
    </para>
    <para>
The solution to this problem is to use a reference count: everyone who has a pointer to the object increases it when they first get the object, and drops the reference count when they're finished with it. Whoever drops it to zero knows it is unused, and can actually delete it.
    </para>
    <para>
Here is the code:
    </para>

<programlisting>

```
--- cache.c.interrupt   2003-12-09 14:25:43.000000000 +1100
+++ cache.c.refcnt      2003-12-09 14:33:05.000000000 +1100
@@ -7,6 +7,7 @@
 struct object
 {
         struct list_head list;
+        unsigned int refcnt;
         int id;
         char name[32];
         int popularity;
@@ -17,6 +18,35 @@
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

+static void __object_put(struct object *obj)
+{
+        if (--obj-&gt;refcnt == 0)
+                kfree(obj);
+}
+
+static void __object_get(struct object *obj)
+{
+        obj-&gt;refcnt++;
+}
+
+void object_put(struct object *obj)
+{
```

```
+        unsigned long flags;
+
+        spin_lock_irqsave(&amp;cache_lock, flags);
+        __object_put(obj);
+        spin_unlock_irqrestore(&amp;cache_lock, flags);
+}
+
+void object_get(struct object *obj)
+{
+        unsigned long flags;
+
+        spin_lock_irqsave(&amp;cache_lock, flags);
+        __object_get(obj);
+        spin_unlock_irqrestore(&amp;cache_lock, flags);
+}
+
 /* Must be holding cache_lock */
 static struct object *__cache_find(int id)
 {
@@ -35,6 +65,7 @@
 {
         BUG_ON(!obj);
         list_del(&amp;obj-&gt;list);
+        __object_put(obj);
         cache_num--;
 }

@@ -63,6 +94,7 @@
         strlcpy(obj-&gt;name, name, sizeof(obj-&gt;name));
         obj-&gt;id = id;
         obj-&gt;popularity = 0;
+        obj-&gt;refcnt = 1; /* The cache holds a reference */

         spin_lock_irqsave(&amp;cache_lock, flags);
         __cache_add(obj);
@@ -79,18 +111,15 @@
         spin_unlock_irqrestore(&amp;cache_lock, flags);
 }

-int cache_find(int id, char *name)
+struct object *cache_find(int id)
 {
         struct object *obj;
-        int ret = -ENOENT;
         unsigned long flags;

         spin_lock_irqsave(&amp;cache_lock, flags);
         obj = __cache_find(id);
-        if (obj) {
-                ret = 0;
-                strcpy(name, obj-&gt;name);
-        }
+        if (obj)
+                __object_get(obj);
         spin_unlock_irqrestore(&amp;cache_lock, flags);
-        return ret;
```

```
+        return obj;
 }
```

</programlisting>

<para>
We encapsulate the reference counting in the standard 'get' and 'put' functions. Now we can return the object itself from <function>cache_find</function> which has the advantage that the user can now sleep holding the object (eg. to <function>copy_to_user</function> to name to userspace).
</para>
<para>
The other point to note is that I said a reference should be held for every pointer to the object: thus the reference count is 1 when first inserted into the cache. In some versions the framework does not hold a reference count, but they are more complicated.
</para>

<sect2 id="examples-refcnt-atomic">
<title>Using Atomic Operations For The Reference Count</title>
<para>
In practice, <type>atomic_t</type> would usually be used for <structfield>refcnt</structfield>. There are a number of atomic operations defined in

<filename class="headerfile">include/asm/atomic.h</filename>: these are guaranteed to be seen atomically from all CPUs in the system, so no lock is required. In this case, it is simpler than using spinlocks, although for anything non-trivial using spinlocks is clearer. The <function>atomic_inc</function> and <function>atomic_dec_and_test</function> are used instead of the standard increment and decrement operators, and the lock is no longer used to protect the reference count itself.
</para>

<programlisting>

```
--- cache.c.refcnt      2003-12-09 15:00:35.000000000 +1100
+++ cache.c.refcnt-atomic       2003-12-11 15:49:42.000000000 +1100
@@ -7,7 +7,7 @@
 struct object
 {
         struct list_head list;
-        unsigned int refcnt;
+        atomic_t refcnt;
         int id;
         char name[32];
         int popularity;
@@ -18,33 +18,15 @@
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

-static void __object_put(struct object *obj)
-{
-        if (--obj-&gt;refcnt == 0)
-                kfree(obj);
-}
```

```
-
-static void __object_get(struct object *obj)
-{
-        obj-&gt;refcnt++;
-}
-
 void object_put(struct object *obj)
 {
-        unsigned long flags;
-
-        spin_lock_irqsave(&amp;cache_lock, flags);
-        __object_put(obj);
-        spin_unlock_irqrestore(&amp;cache_lock, flags);
+        if (atomic_dec_and_test(&amp;obj-&gt;refcnt))
+                kfree(obj);
 }

 void object_get(struct object *obj)
 {
-        unsigned long flags;
-
-        spin_lock_irqsave(&amp;cache_lock, flags);
-        __object_get(obj);
-        spin_unlock_irqrestore(&amp;cache_lock, flags);
+        atomic_inc(&amp;obj-&gt;refcnt);
 }

 /* Must be holding cache_lock */
@@ -65,7 +47,7 @@
 {
         BUG_ON(!obj);
         list_del(&amp;obj-&gt;list);
-        __object_put(obj);
+        object_put(obj);
         cache_num--;
 }

@@ -94,7 +76,7 @@
         strlcpy(obj-&gt;name, name, sizeof(obj-&gt;name));
         obj-&gt;id = id;
         obj-&gt;popularity = 0;
-        obj-&gt;refcnt = 1; /* The cache holds a reference */
+        atomic_set(&amp;obj-&gt;refcnt, 1); /* The cache holds a reference */

         spin_lock_irqsave(&amp;cache_lock, flags);
         __cache_add(obj);
@@ -119,7 +101,7 @@
         spin_lock_irqsave(&amp;cache_lock, flags);
         obj = __cache_find(id);
         if (obj)
-                __object_get(obj);
+                object_get(obj);
         spin_unlock_irqrestore(&amp;cache_lock, flags);
         return obj;
 }
</programlisting>
```

```
</sect2>
</sect1>

   <sect1 id="examples-lock-per-obj">
    <title>Protecting The Objects Themselves</title>
    <para>
In these examples, we assumed that the objects (except the reference
counts) never changed once they are created.  If we wanted to allow
the name to change, there are three possibilities:
    </para>
    <itemizedlist>
      <listitem>
        <para>
You can make <symbol>cache_lock</symbol> non-static, and tell people
to grab that lock before changing the name in any object.
        </para>
      </listitem>
      <listitem>
        <para>
You can provide a <function>cache_obj_rename</function> which grabs
this lock and changes the name for the caller, and tell everyone to
use that function.
        </para>
      </listitem>
      <listitem>
        <para>
You can make the <symbol>cache_lock</symbol> protect only the cache
itself, and use another lock to protect the name.
        </para>
      </listitem>
    </itemizedlist>

      <para>
Theoretically, you can make the locks as fine-grained as one lock for
every field, for every object.  In practice, the most common variants
are:
</para>
    <itemizedlist>
      <listitem>
        <para>
One lock which protects the infrastructure (the <symbol>cache</symbol>
list in this example) and all the objects.  This is what we have done
so far.
        </para>
      </listitem>
      <listitem>
        <para>
One lock which protects the infrastructure (including the list
pointers inside the objects), and one lock inside the object which
protects the rest of that object.
        </para>
      </listitem>
      <listitem>
        <para>
Multiple locks to protect the infrastructure (eg. one lock per hash
chain), possibly with a separate per-object lock.
```

```
        </para>
      </listitem>
    </itemizedlist>

<para>
Here is the "lock-per-object" implementation:
</para>
<programlisting>
--- cache.c.refcnt-atomic       2003-12-11 15:50:54.000000000 +1100
+++ cache.c.perobjectlock       2003-12-11 17:15:03.000000000 +1100
@@ -6,11 +6,17 @@

 struct object
 {
+        /* These two protected by cache_lock. */
         struct list_head list;
+        int popularity;
+
         atomic_t refcnt;
+
+        /* Doesn't change once created. */
         int id;
+
+        spinlock_t lock; /* Protects the name */
         char name[32];
-        int popularity;
 };

 static DEFINE_SPINLOCK(cache_lock);
@@ -77,6 +84,7 @@
         obj-&gt;id = id;
         obj-&gt;popularity = 0;
         atomic_set(&amp;obj-&gt;refcnt, 1); /* The cache holds a reference */
+        spin_lock_init(&amp;obj-&gt;lock);

         spin_lock_irqsave(&amp;cache_lock, flags);
         __cache_add(obj);
</programlisting>

<para>
Note that I decide that the <structfield>popularity</structfield>
count should be protected by the <symbol>cache_lock</symbol> rather
than the per-object lock: this is because it (like the
<structname>struct list_head</structname> inside the object) is
logically part of the infrastructure.  This way, I don't need to grab
the lock of every object in <function>__cache_add</function> when
seeking the least popular.
</para>

<para>
I also decided that the <structfield>id</structfield> member is
unchangeable, so I don't need to grab each object lock in
<function>__cache_find()</function> to examine the
<structfield>id</structfield>: the object lock is only used by a
caller who wants to read or write the <structfield>name</structfield>
field.
```

```
</para>

<para>
Note also that I added a comment describing what data was protected by
which locks.  This is extremely important, as it describes the runtime
behavior of the code, and can be hard to gain from just reading.  And
as Alan Cox says, <quote>Lock data, not code</quote>.
</para>
</sect1>
</chapter>

   <chapter id="common-problems">
    <title>Common Problems</title>
    <sect1 id="deadlock">
    <title>Deadlock: Simple and Advanced</title>

    <para>
      There is a coding bug where a piece of code tries to grab a
      spinlock twice: it will spin forever, waiting for the lock to
      be released (spinlocks, rwlocks and mutexes are not
      recursive in Linux).  This is trivial to diagnose: not a
      stay-up-five-nights-talk-to-fluffy-code-bunnies kind of
      problem.
    </para>

    <para>
      For a slightly more complex case, imagine you have a region
      shared by a softirq and user context.  If you use a
      <function>spin_lock()</function> call to protect it, it is
      possible that the user context will be interrupted by the softirq
      while it holds the lock, and the softirq will then spin
      forever trying to get the same lock.
    </para>

    <para>
      Both of these are called deadlock, and as shown above, it can
      occur even with a single CPU (although not on UP compiles,
      since spinlocks vanish on kernel compiles with
      <symbol>CONFIG_SMP</symbol>=n. You'll still get data corruption
      in the second example).
    </para>

    <para>
      This complete lockup is easy to diagnose: on SMP boxes the
      watchdog timer or compiling with <symbol>DEBUG_SPINLOCK</symbol> set
      (<filename>include/linux/spinlock.h</filename>) will show this up
      immediately when it happens.
    </para>

    <para>
      A more complex problem is the so-called 'deadly embrace',
      involving two or more locks.  Say you have a hash table: each
      entry in the table is a spinlock, and a chain of hashed
      objects.  Inside a softirq handler, you sometimes want to
      alter an object from one place in the hash to another: you
      grab the spinlock of the old hash chain and the spinlock of
```

```
  the new hash chain, and delete the object from the old one,
  and insert it in the new one.
</para>

<para>
  There are two problems here.  First, if your code ever
  tries to move the object to the same chain, it will deadlock
  with itself as it tries to lock it twice.  Secondly, if the
  same softirq on another CPU is trying to move another object
  in the reverse direction, the following could happen:
</para>

<table>
 <title>Consequences</title>

 <tgroup cols="2" align="left">

  <thead>
   <row>
    <entry>CPU 1</entry>
    <entry>CPU 2</entry>
   </row>
  </thead>

  <tbody>
   <row>
    <entry>Grab lock A -&gt; OK</entry>
    <entry>Grab lock B -&gt; OK</entry>
   </row>
   <row>
    <entry>Grab lock B -&gt; spin</entry>
    <entry>Grab lock A -&gt; spin</entry>
   </row>
  </tbody>
 </tgroup>
</table>

<para>
  The two CPUs will spin forever, waiting for the other to give up
  their lock.  It will look, smell, and feel like a crash.
</para>
</sect1>

<sect1 id="techs-deadlock-prevent">
 <title>Preventing Deadlock</title>

 <para>
   Textbooks will tell you that if you always lock in the same
   order, you will never get this kind of deadlock.  Practice
   will tell you that this approach doesn't scale: when I
   create a new lock, I don't understand enough of the kernel
   to figure out where in the 5000 lock hierarchy it will fit.
 </para>

 <para>
   The best locks are encapsulated: they never get exposed in
```

```
    headers, and are never held around calls to non-trivial
    functions outside the same file.  You can read through this
    code and see that it will never deadlock, because it never
    tries to grab another lock while it has that one.  People
    using your code don't even need to know you are using a
    lock.
  </para>

  <para>
    A classic problem here is when you provide callbacks or
    hooks: if you call these with the lock held, you risk simple
    deadlock, or a deadly embrace (who knows what the callback
    will do?).  Remember, the other programmers are out to get
    you, so don't do this.
  </para>

 <sect2 id="techs-deadlock-overprevent">
  <title>Overzealous Prevention Of Deadlocks</title>

  <para>
    Deadlocks are problematic, but not as bad as data
    corruption.  Code which grabs a read lock, searches a list,
    fails to find what it wants, drops the read lock, grabs a
    write lock and inserts the object has a race condition.
  </para>

  <para>
    If you don't see why, please stay the fuck away from my code.
  </para>
 </sect2>
 </sect1>

<sect1 id="racing-timers">
 <title>Racing Timers: A Kernel Pastime</title>

 <para>
   Timers can produce their own special problems with races.
   Consider a collection of objects (list, hash, etc) where each
   object has a timer which is due to destroy it.
 </para>

 <para>
   If you want to destroy the entire collection (say on module
   removal), you might do the following:
 </para>

 <programlisting>
        /* THIS CODE BAD BAD BAD BAD: IF IT WAS ANY WORSE IT WOULD USE
           HUNGARIAN NOTATION */
        spin_lock_bh(&amp;list_lock);

        while (list) {
                struct foo *next = list-&gt;next;
                del_timer(&amp;list-&gt;timer);
                kfree(list);
                list = next;
```

```
        }

        spin_unlock_bh(&amp;list_lock);
    </programlisting>

    <para>
      Sooner or later, this will crash on SMP, because a timer can
      have just gone off before the <function>spin_lock_bh()</function>,
      and it will only get the lock after we
      <function>spin_unlock_bh()</function>, and then try to free
      the element (which has already been freed!).
    </para>

    <para>
      This can be avoided by checking the result of
      <function>del_timer()</function>: if it returns
      <returnvalue>1</returnvalue>, the timer has been deleted.
      If <returnvalue>0</returnvalue>, it means (in this
      case) that it is currently running, so we can do:
    </para>

    <programlisting>
        retry:
                spin_lock_bh(&amp;list_lock);

                while (list) {
                        struct foo *next = list-&gt;next;
                        if (!del_timer(&amp;list-&gt;timer)) {
                                /* Give timer a chance to delete this */
                                spin_unlock_bh(&amp;list_lock);
                                goto retry;
                        }
                        kfree(list);
                        list = next;
                }

                spin_unlock_bh(&amp;list_lock);
    </programlisting>

    <para>
      Another common problem is deleting timers which restart
      themselves (by calling <function>add_timer()</function> at the end
      of their timer function).  Because this is a fairly common case
      which is prone to races, you should use
<function>del_timer_sync()</function>
      (<filename class="headerfile">include/linux/timer.h</filename>)
      to handle this case.  It returns the number of times the timer
      had to be deleted before we finally stopped it from adding itself back
      in.
    </para>
   </sect1>

  </chapter>

 <chapter id="Efficiency">
    <title>Locking Speed</title>
```

```
    <para>
There are three main things to worry about when considering speed of
some code which does locking.  First is concurrency: how many things
are going to be waiting while someone else is holding a lock.  Second
is the time taken to actually acquire and release an uncontended lock.
Third is using fewer, or smarter locks.  I'm assuming that the lock is
used fairly often: otherwise, you wouldn't be concerned about
efficiency.
</para>
    <para>
Concurrency depends on how long the lock is usually held: you should
hold the lock for as long as needed, but no longer.  In the cache
example, we always create the object without the lock held, and then
grab the lock only when we are ready to insert it in the list.
</para>
    <para>
Acquisition times depend on how much damage the lock operations do to
the pipeline (pipeline stalls) and how likely it is that this CPU was
the last one to grab the lock (ie. is the lock cache-hot for this
CPU): on a machine with more CPUs, this likelihood drops fast.
Consider a 700MHz Intel Pentium III: an instruction takes about 0.7ns,
an atomic increment takes about 58ns, a lock which is cache-hot on
this CPU takes 160ns, and a cacheline transfer from another CPU takes
an additional 170 to 360ns.  (These figures from Paul McKenney's
<ulink url="http://www.linuxjournal.com/article.php?sid=6993"> Linux
Journal RCU article</ulink>).
</para>
    <para>
These two aims conflict: holding a lock for a short time might be done
by splitting locks into parts (such as in our final per-object-lock
example), but this increases the number of lock acquisitions, and the
results are often slower than having a single lock.  This is another
reason to advocate locking simplicity.
</para>
    <para>
The third concern is addressed below: there are some methods to reduce
the amount of locking which needs to be done.
</para>

  <sect1 id="efficiency-rwlocks">
   <title>Read/Write Lock Variants</title>

   <para>
      Both spinlocks and mutexes have read/write variants:
      <type>rwlock_t</type> and <structname>struct rw_semaphore</structname>.
      These divide users into two classes: the readers and the writers.  If
      you are only reading the data, you can get a read lock, but to write to
      the data you need the write lock.  Many people can hold a read lock,
      but a writer must be sole holder.
    </para>

   <para>
      If your code divides neatly along reader/writer lines (as our
      cache code does), and the lock is held by readers for
      significant lengths of time, using these locks can help.  They
```

```
   are slightly slower than the normal locks though, so in practice
   <type>rwlock_t</type> is not usually worthwhile.
  </para>
 </sect1>

 <sect1 id="efficiency-read-copy-update">
  <title>Avoiding Locks: Read Copy Update</title>

  <para>
   There is a special method of read/write locking called Read Copy
   Update.  Using RCU, the readers can avoid taking a lock
   altogether: as we expect our cache to be read more often than
   updated (otherwise the cache is a waste of time), it is a
   candidate for this optimization.
  </para>

  <para>
   How do we get rid of read locks?  Getting rid of read locks
   means that writers may be changing the list underneath the
   readers.  That is actually quite simple: we can read a linked
   list while an element is being added if the writer adds the
   element very carefully.  For example, adding
   <symbol>new</symbol> to a single linked list called
   <symbol>list</symbol>:
  </para>

  <programlisting>
     new-&gt;next = list-&gt;next;
     wmb();
     list-&gt;next = new;
  </programlisting>

  <para>
   The <function>wmb()</function> is a write memory barrier.  It
   ensures that the first operation (setting the new element's
   <symbol>next</symbol> pointer) is complete and will be seen by
   all CPUs, before the second operation is (putting the new
   element into the list).  This is important, since modern
   compilers and modern CPUs can both reorder instructions unless
   told otherwise: we want a reader to either not see the new
   element at all, or see the new element with the
   <symbol>next</symbol> pointer correctly pointing at the rest of
   the list.
  </para>
  <para>
   Fortunately, there is a function to do this for standard
   <structname>struct list_head</structname> lists:
   <function>list_add_rcu()</function>
   (<filename>include/linux/list.h</filename>).
  </para>
  <para>
   Removing an element from the list is even simpler: we replace
   the pointer to the old element with a pointer to its successor,
   and readers will either see it, or skip over it.
  </para>
  <programlisting>
```

```
        list-&gt;next = old-&gt;next;
    </programlisting>
    <para>
      There is <function>list_del_rcu()</function>
      (<filename>include/linux/list.h</filename>) which does this (the
      normal version poisons the old object, which we don't want).
    </para>
    <para>
      The reader must also be careful: some CPUs can look through the
      <symbol>next</symbol> pointer to start reading the contents of
      the next element early, but don't realize that the pre-fetched
      contents is wrong when the <symbol>next</symbol> pointer changes
      underneath them.  Once again, there is a
      <function>list_for_each_entry_rcu()</function>
      (<filename>include/linux/list.h</filename>) to help you.  Of
      course, writers can just use
      <function>list_for_each_entry()</function>, since there cannot
      be two simultaneous writers.
    </para>
    <para>
      Our final dilemma is this: when can we actually destroy the
      removed element?  Remember, a reader might be stepping through
      this element in the list right now: if we free this element and
      the <symbol>next</symbol> pointer changes, the reader will jump
      off into garbage and crash.  We need to wait until we know that
      all the readers who were traversing the list when we deleted the
      element are finished.  We use <function>call_rcu()</function> to
      register a callback which will actually destroy the object once
      the readers are finished.
    </para>
    <para>
      But how does Read Copy Update know when the readers are
      finished?  The method is this: firstly, the readers always
      traverse the list inside

<function>rcu_read_lock()</function>/<function>rcu_read_unlock()</function>
      pairs: these simply disable preemption so the reader won't go to
      sleep while reading the list.
    </para>
    <para>
      RCU then waits until every other CPU has slept at least once:
      since readers cannot sleep, we know that any readers which were
      traversing the list during the deletion are finished, and the
      callback is triggered.  The real Read Copy Update code is a
      little more optimized than this, but this is the fundamental
      idea.
    </para>

<programlisting>
--- cache.c.perobjectlock       2003-12-11 17:15:03.000000000 +1100
+++ cache.c.rcupdate    2003-12-11 17:55:14.000000000 +1100
@@ -1,15 +1,18 @@
 #include &lt;linux/list.h&gt;
 #include &lt;linux/slab.h&gt;
 #include &lt;linux/string.h&gt;
+#include &lt;linux/rcupdate.h&gt;
```

```
 #include &lt;linux/mutex.h&gt;
 #include &lt;asm/errno.h&gt;

 struct object
 {
-        /* These two protected by cache_lock. */
+        /* This is protected by RCU */
         struct list_head list;
         int popularity;

+        struct rcu_head rcu;
+
         atomic_t refcnt;

         /* Doesn't change once created. */
@@ -40,7 +43,7 @@
 {
         struct object *i;

-        list_for_each_entry(i, &amp;cache, list) {
+        list_for_each_entry_rcu(i, &amp;cache, list) {
                 if (i-&gt;id == id) {
                         i-&gt;popularity++;
                         return i;
@@ -49,19 +52,25 @@
         return NULL;
 }

+/* Final discard done once we know no readers are looking. */
+static void cache_delete_rcu(void *arg)
+{
+        object_put(arg);
+}
+
 /* Must be holding cache_lock */
 static void __cache_delete(struct object *obj)
 {
         BUG_ON(!obj);
-        list_del(&amp;obj-&gt;list);
-        object_put(obj);
+        list_del_rcu(&amp;obj-&gt;list);
         cache_num--;
+        call_rcu(&amp;obj-&gt;rcu, cache_delete_rcu, obj);
 }

 /* Must be holding cache_lock */
 static void __cache_add(struct object *obj)
 {
-        list_add(&amp;obj-&gt;list, &amp;cache);
+        list_add_rcu(&amp;obj-&gt;list, &amp;cache);
         if (++cache_num > MAX_CACHE_SIZE) {
                 struct object *i, *outcast = NULL;
                 list_for_each_entry(i, &amp;cache, list) {
@@ -85,6 +94,7 @@
         obj-&gt;popularity = 0;
         atomic_set(&amp;obj-&gt;refcnt, 1); /* The cache holds a reference */
```

```
         spin_lock_init(&amp;obj-&gt;lock);
+        INIT_RCU_HEAD(&amp;obj-&gt;rcu);

         spin_lock_irqsave(&amp;cache_lock, flags);
         __cache_add(obj);
@@ -104,12 +114,11 @@
 struct object *cache_find(int id)
 {
         struct object *obj;
-        unsigned long flags;

-        spin_lock_irqsave(&amp;cache_lock, flags);
+        rcu_read_lock();
         obj = __cache_find(id);
         if (obj)
                 object_get(obj);
-        spin_unlock_irqrestore(&amp;cache_lock, flags);
+        rcu_read_unlock();
         return obj;
 }
</programlisting>

<para>
Note that the reader will alter the
<structfield>popularity</structfield> member in
<function>__cache_find()</function>, and now it doesn't hold a lock.
One solution would be to make it an <type>atomic_t</type>, but for
this usage, we don't really care about races: an approximate result is
good enough, so I didn't change it.
</para>

<para>
The result is that <function>cache_find()</function> requires no
synchronization with any other functions, so is almost as fast on SMP
as it would be on UP.
</para>

<para>
There is a furthur optimization possible here: remember our original
cache code, where there were no reference counts and the caller simply
held the lock whenever using the object?  This is still possible: if
you hold the lock, noone can delete the object, so you don't need to
get and put the reference count.
</para>

<para>
Now, because the 'read lock' in RCU is simply disabling preemption, a
caller which always has preemption disabled between calling
<function>cache_find()</function> and
<function>object_put()</function> does not need to actually get and
put the reference count: we could expose
<function>__cache_find()</function> by making it non-static, and
such callers could simply call that.
</para>
<para>
The benefit here is that the reference count is not written to: the
```

```
object is not altered in any way, which is much faster on SMP
machines due to caching.
</para>
  </sect1>

   <sect1 id="per-cpu">
    <title>Per-CPU Data</title>

    <para>
      Another technique for avoiding locking which is used fairly
      widely is to duplicate information for each CPU.  For example,
      if you wanted to keep a count of a common condition, you could
      use a spin lock and a single counter.  Nice and simple.
    </para>

    <para>
      If that was too slow (it's usually not, but if you've got a
      really big machine to test on and can show that it is), you
      could instead use a counter for each CPU, then none of them need
      an exclusive lock.  See <function>DEFINE_PER_CPU()</function>,
      <function>get_cpu_var()</function> and
      <function>put_cpu_var()</function>
      (<filename class="headerfile">include/linux/percpu.h</filename>).
    </para>

    <para>
      Of particular use for simple per-cpu counters is the
      <type>local_t</type> type, and the
      <function>cpu_local_inc()</function> and related functions,
      which are more efficient than simple code on some architectures
      (<filename class="headerfile">include/asm/local.h</filename>).
    </para>

    <para>
      Note that there is no simple, reliable way of getting an exact
      value of such a counter, without introducing more locks.  This
      is not a problem for some uses.
    </para>
   </sect1>

   <sect1 id="mostly-hardirq">
    <title>Data Which Mostly Used By An IRQ Handler</title>

    <para>
      If data is always accessed from within the same IRQ handler, you
      don't need a lock at all: the kernel already guarantees that the
      irq handler will not run simultaneously on multiple CPUs.
    </para>
    <para>
      Manfred Spraul points out that you can still do this, even if
      the data is very occasionally accessed in user context or
      softirqs/tasklets.  The irq handler doesn't use a lock, and
      all other accesses are done as so:
    </para>

<programlisting>
```

```
        spin_lock(&amp;lock);
        disable_irq(irq);
        ...
        enable_irq(irq);
        spin_unlock(&amp;lock);
</programlisting>
    <para>
      The <function>disable_irq()</function> prevents the irq handler
      from running (and waits for it to finish if it's currently
      running on other CPUs).  The spinlock prevents any other
      accesses happening at the same time.  Naturally, this is slower
      than just a <function>spin_lock_irq()</function> call, so it
      only makes sense if this type of access happens extremely
      rarely.
    </para>
   </sect1>
  </chapter>

 <chapter id="sleeping-things">
    <title>What Functions Are Safe To Call From Interrupts?</title>

    <para>
      Many functions in the kernel sleep (ie. call schedule())
      directly or indirectly: you can never call them while holding a
      spinlock, or with preemption disabled.  This also means you need
      to be in user context: calling them from an interrupt is illegal.
    </para>

   <sect1 id="sleeping">
    <title>Some Functions Which Sleep</title>

    <para>
      The most common ones are listed below, but you usually have to
      read the code to find out if other calls are safe.  If everyone
      else who calls it can sleep, you probably need to be able to
      sleep, too.  In particular, registration and deregistration
      functions usually expect to be called from user context, and can
      sleep.
    </para>

    <itemizedlist>
     <listitem>
      <para>
        Accesses to
        <firstterm linkend="gloss-userspace">userspace</firstterm>:
      </para>
      <itemizedlist>
       <listitem>
        <para>
          <function>copy_from_user()</function>
        </para>
       </listitem>
       <listitem>
        <para>
          <function>copy_to_user()</function>
        </para>
```

```
    </listitem>
    <listitem>
     <para>
       <function>get_user()</function>
     </para>
    </listitem>
    <listitem>
     <para>
       <function>put_user()</function>
     </para>
    </listitem>
   </itemizedlist>
  </listitem>

  <listitem>
   <para>
     <function>kmalloc(GFP_KERNEL)</function>
   </para>
  </listitem>

  <listitem>
   <para>
   <function>mutex_lock_interruptible()</function> and
   <function>mutex_lock()</function>
   </para>
   <para>
    There is a <function>mutex_trylock()</function> which can be
    used inside interrupt context, as it will not sleep.
    <function>mutex_unlock()</function> will also never sleep.
   </para>
  </listitem>
 </itemizedlist>
</sect1>

<sect1 id="dont-sleep">
 <title>Some Functions Which Don't Sleep</title>

 <para>
  Some functions are safe to call from any context, or holding
  almost any lock.
 </para>

 <itemizedlist>
  <listitem>
   <para>
     <function>printk()</function>
   </para>
  </listitem>
  <listitem>
   <para>
     <function>kfree()</function>
   </para>
  </listitem>
  <listitem>
   <para>
     <function>add_timer()</function> and <function>del_timer()</function>
```

```
    </para>
   </listitem>
  </itemizedlist>
 </sect1>
</chapter>

<chapter id="references">
 <title>Further reading</title>

 <itemizedlist>
  <listitem>
   <para>
     <filename>Documentation/spinlocks.txt</filename>:
     Linus Torvalds' spinlocking tutorial in the kernel sources.
   </para>
  </listitem>

  <listitem>
   <para>
     Unix Systems for Modern Architectures: Symmetric
     Multiprocessing and Caching for Kernel Programmers:
   </para>

   <para>
     Curt Schimmel's very good introduction to kernel level
     locking (not written for Linux, but nearly everything
     applies).  The book is expensive, but really worth every
     penny to understand SMP locking. [ISBN: 0201633388]
   </para>
  </listitem>
 </itemizedlist>
</chapter>

<chapter id="thanks">
  <title>Thanks</title>

  <para>
    Thanks to Telsa Gwynne for DocBooking, neatening and adding
    style.
  </para>

  <para>
    Thanks to Martin Pool, Philipp Rumpf, Stephen Rothwell, Paul
    Mackerras, Ruedi Aschwanden, Alan Cox, Manfred Spraul, Tim
    Waugh, Pete Zaitcev, James Morris, Robert Love, Paul McKenney,
    John Ashby for proofreading, correcting, flaming, commenting.
  </para>

  <para>
    Thanks to the cabal for having no influence on this document.
  </para>
</chapter>

<glossary id="glossary">
 <title>Glossary</title>
```

```
<glossentry id="gloss-preemption">
 <glossterm>preemption</glossterm>
  <glossdef>
   <para>
     Prior to 2.5, or when <symbol>CONFIG_PREEMPT</symbol> is
     unset, processes in user context inside the kernel would not
     preempt each other (ie. you had that CPU until you gave it up,
     except for interrupts).  With the addition of
     <symbol>CONFIG_PREEMPT</symbol> in 2.5.4, this changed: when
     in user context, higher priority tasks can "cut in": spinlocks
     were changed to disable preemption, even on UP.
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-bh">
 <glossterm>bh</glossterm>
  <glossdef>
   <para>
     Bottom Half: for historical reasons, functions with
     '_bh' in them often now refer to any software interrupt, e.g.
     <function>spin_lock_bh()</function> blocks any software interrupt
     on the current CPU.  Bottom halves are deprecated, and will
     eventually be replaced by tasklets.  Only one bottom half will be
     running at any time.
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-hwinterrupt">
 <glossterm>Hardware Interrupt / Hardware IRQ</glossterm>
 <glossdef>
  <para>
    Hardware interrupt request.  <function>in_irq()</function> returns
    <returnvalue>true</returnvalue> in a hardware interrupt handler.
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-interruptcontext">
 <glossterm>Interrupt Context</glossterm>
 <glossdef>
  <para>
    Not user context: processing a hardware irq or software irq.
    Indicated by the <function>in_interrupt()</function> macro
    returning <returnvalue>true</returnvalue>.
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-smp">
 <glossterm><acronym>SMP</acronym></glossterm>
 <glossdef>
  <para>
    Symmetric Multi-Processor: kernels compiled for multiple-CPU
    machines.  (CONFIG_SMP=y).
```

```
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-softirq">
 <glossterm>Software Interrupt / softirq</glossterm>
 <glossdef>
  <para>
    Software interrupt handler.  <function>in_irq()</function> returns
    <returnvalue>false</returnvalue>; <function>in_softirq()</function>
    returns <returnvalue>true</returnvalue>.  Tasklets and softirqs
     both fall into the category of 'software interrupts'.
  </para>
  <para>
    Strictly speaking a softirq is one of up to 32 enumerated software
    interrupts which can run on multiple CPUs at once.
    Sometimes used to refer to tasklets as
    well (ie. all software interrupts).
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-tasklet">
 <glossterm>tasklet</glossterm>
 <glossdef>
  <para>
    A dynamically-registrable software interrupt,
    which is guaranteed to only run on one CPU at a time.
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-timers">
 <glossterm>timer</glossterm>
 <glossdef>
  <para>
    A dynamically-registrable software interrupt, which is run at
    (or close to) a given time.  When running, it is just like a
    tasklet (in fact, they are called from the TIMER_SOFTIRQ).
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-up">
 <glossterm><acronym>UP</acronym></glossterm>
 <glossdef>
  <para>
    Uni-Processor: Non-SMP.  (CONFIG_SMP=n).
  </para>
 </glossdef>
</glossentry>

<glossentry id="gloss-usercontext">
 <glossterm>User Context</glossterm>
 <glossdef>
  <para>
```

```
        The kernel executing on behalf of a particular process (ie. a
        system call or trap) or kernel thread.  You can tell which
        process with the <symbol>current</symbol> macro.)  Not to
        be confused with userspace.  Can be interrupted by software or
        hardware interrupts.
      </para>
     </glossdef>
    </glossentry>

    <glossentry id="gloss-userspace">
     <glossterm>Userspace</glossterm>
     <glossdef>
      <para>
        A process executing its own code outside the kernel.
      </para>
     </glossdef>
    </glossentry>

  </glossary>
</book>
```