```
<?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="lk-hacking-guide">
 <bookinfo>
  <title>Unreliable Guide To Hacking The Linux Kernel</title>

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

  <copyright>
   <year>2005</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>

  <releaseinfo>
   This is the first release of this document as part of the kernel tarball.
```

```
 </releaseinfo>

</bookinfo>

<toc></toc>

<chapter id="introduction">
 <title>Introduction</title>
 <para>
  Welcome, gentle reader, to Rusty's Remarkably Unreliable Guide to Linux
  Kernel Hacking.  This document describes the common routines and
  general requirements for kernel code: its goal is to serve as a
  primer for Linux kernel development for experienced C
  programmers.  I avoid implementation details: that's what the
  code is for, and I ignore whole tracts of useful routines.
 </para>
 <para>
  Before you read this, please understand that I never wanted to
  write this document, being grossly under-qualified, but I always
  wanted to read it, and this was the only way.  I hope it will
  grow into a compendium of best practice, common starting points
  and random information.
 </para>
</chapter>

<chapter id="basic-players">
 <title>The Players</title>

 <para>
  At any time each of the CPUs in a system can be:
 </para>

 <itemizedlist>
  <listitem>
   <para>
    not associated with any process, serving a hardware interrupt;
   </para>
  </listitem>

  <listitem>
   <para>
    not associated with any process, serving a softirq or tasklet;
   </para>
  </listitem>

  <listitem>
   <para>
    running in kernel space, associated with a process (user context);
   </para>
  </listitem>

  <listitem>
   <para>
    running a process in user space.
   </para>
  </listitem>
```

</itemizedlist>

<para>
 There is an ordering between these. The bottom two can preempt
 each other, but above that is a strict hierarchy: each can only be
 preempted by the ones above it. For example, while a softirq is
 running on a CPU, no other softirq will preempt it, but a hardware
 interrupt can. However, any other CPUs in the system execute
 independently.
</para>

<para>
 We'll see a number of ways that the user context can block
 interrupts, to become truly non-preemptable.
</para>

<sect1 id="basics-usercontext">
 <title>User Context</title>

 <para>
  User context is when you are coming in from a system call or other
  trap: like userspace, you can be preempted by more important tasks
  and by interrupts. You can sleep, by calling
  <function>schedule()</function>.
 </para>

 <note>
  <para>
   You are always in user context on module load and unload,
   and on operations on the block device layer.
  </para>
 </note>

 <para>
  In user context, the <varname>current</varname> pointer (indicating
  the task we are currently executing) is valid, and
  <function>in_interrupt()</function>
  (<filename>include/linux/interrupt.h</filename>) is <returnvalue>false
  </returnvalue>.
 </para>

 <caution>
  <para>
   Beware that if you have preemption or softirqs disabled
   (see below), <function>in_interrupt()</function> will return a
   false positive.
  </para>
 </caution>
</sect1>

<sect1 id="basics-hardirqs">
 <title>Hardware Interrupts (Hard IRQs)</title>

 <para>
  Timer ticks, <hardware>network cards</hardware> and
  <hardware>keyboard</hardware> are examples of real

hardware which produce interrupts at any time. The kernel runs
interrupt handlers, which services the hardware. The kernel
guarantees that this handler is never re-entered: if the same
interrupt arrives, it is queued (or dropped). Because it
disables interrupts, this handler has to be fast: frequently it
simply acknowledges the interrupt, marks a 'software interrupt'
for execution and exits.
</para>

<para>
You can tell you are in a hardware interrupt, because
<function>in_irq()</function> returns <returnvalue>true</returnvalue>.
</para>
<caution>
<para>
Beware that this will return a false positive if interrupts are disabled
(see below).
</para>
</caution>
</sect1>

<sect1 id="basics-softirqs">
<title>Software Interrupt Context: Softirqs and Tasklets</title>

<para>
Whenever a system call is about to return to userspace, or a
hardware interrupt handler exits, any 'software interrupts'
which are marked pending (usually by hardware interrupts) are
run (<filename>kernel/softirq.c</filename>).
</para>

<para>
Much of the real interrupt handling work is done here. Early in
the transition to <acronym>SMP</acronym>, there were only 'bottom
halves' (BHs), which didn't take advantage of multiple CPUs. Shortly
after we switched from wind-up computers made of match-sticks and snot,
we abandoned this limitation and switched to 'softirqs'.
</para>

<para>
<filename class="headerfile">include/linux/interrupt.h</filename> lists the
different softirqs. A very important softirq is the
timer softirq (<filename
class="headerfile">include/linux/timer.h</filename>): you can
register to have it call functions for you in a given length of
time.
</para>

<para>
Softirqs are often a pain to deal with, since the same softirq
will run simultaneously on more than one CPU. For this reason,
tasklets (<filename
class="headerfile">include/linux/interrupt.h</filename>) are more
often used: they are dynamically-registrable (meaning you can have
as many as you want), and they also guarantee that any tasklet
will only run on one CPU at any time, although different tasklets

```
   can run simultaneously.
  </para>
  <caution>
   <para>
    The name 'tasklet' is misleading: they have nothing to do with 'tasks',
    and probably more to do with some bad vodka Alexey Kuznetsov had at the
    time.
   </para>
  </caution>

  <para>
   You can tell you are in a softirq (or tasklet)
   using the <function>in_softirq()</function> macro
   (<filename class="headerfile">include/linux/interrupt.h</filename>).
  </para>
  <caution>
   <para>
    Beware that this will return a false positive if a bh lock (see below)
    is held.
   </para>
  </caution>
 </sect1>
</chapter>

<chapter id="basic-rules">
 <title>Some Basic Rules</title>

 <variablelist>
  <varlistentry>
   <term>No memory protection</term>
   <listitem>
    <para>
     If you corrupt memory, whether in user context or
     interrupt context, the whole machine will crash.  Are you
     sure you can't do what you want in userspace?
    </para>
   </listitem>
  </varlistentry>

  <varlistentry>
   <term>No floating point or <acronym>MMX</acronym></term>
   <listitem>
    <para>
     The <acronym>FPU</acronym> context is not saved; even in user
     context the <acronym>FPU</acronym> state probably won't
     correspond with the current process: you would mess with some
     user process' <acronym>FPU</acronym> state.  If you really want
     to do this, you would have to explicitly save/restore the full
     <acronym>FPU</acronym> state (and avoid context switches).  It
     is generally a bad idea; use fixed point arithmetic first.
    </para>
   </listitem>
  </varlistentry>

  <varlistentry>
   <term>A rigid stack limit</term>
```

```
    <listitem>
     <para>
      Depending on configuration options the kernel stack is about 3K to 6K for
most 32-bit architectures: it's
      about 14K on most 64-bit archs, and often shared with interrupts
      so you can't use it all.  Avoid deep recursion and huge local
      arrays on the stack (allocate them dynamically instead).
     </para>
    </listitem>
   </varlistentry>

   <varlistentry>
    <term>The Linux kernel is portable</term>
    <listitem>
     <para>
      Let's keep it that way.  Your code should be 64-bit clean,
      and endian-independent.  You should also minimize CPU
      specific stuff, e.g. inline assembly should be cleanly
      encapsulated and minimized to ease porting.  Generally it
      should be restricted to the architecture-dependent part of
      the kernel tree.
     </para>
    </listitem>
   </varlistentry>
  </variablelist>
 </chapter>

 <chapter id="ioctls">
  <title>ioctls: Not writing a new system call</title>

  <para>
   A system call generally looks like this
  </para>

  <programlisting>
asmlinkage long sys_mycall(int arg)
{
        return 0;
}
  </programlisting>

  <para>
   First, in most cases you don't want to create a new system call.
   You create a character device and implement an appropriate ioctl
   for it.  This is much more flexible than system calls, doesn't have
   to be entered in every architecture's
   <filename class="headerfile">include/asm/unistd.h</filename> and
   <filename>arch/kernel/entry.S</filename> file, and is much more
   likely to be accepted by Linus.
  </para>

  <para>
   If all your routine does is read or write some parameter, consider
   implementing a <function>sysfs</function> interface instead.
  </para>
```

```
  <para>
   Inside the ioctl you're in user context to a process.  When a
   error occurs you return a negated errno (see
   <filename class="headerfile">include/linux/errno.h</filename>),
   otherwise you return <returnvalue>0</returnvalue>.
  </para>

  <para>
   After you slept you should check if a signal occurred: the
   Unix/Linux way of handling signals is to temporarily exit the
   system call with the <constant>-ERESTARTSYS</constant> error.  The
   system call entry code will switch back to user context, process
   the signal handler and then your system call will be restarted
   (unless the user disabled that).  So you should be prepared to
   process the restart, e.g. if you're in the middle of manipulating
   some data structure.
  </para>

  <programlisting>
if (signal_pending(current))
        return -ERESTARTSYS;
  </programlisting>

  <para>
   If you're doing longer computations: first think userspace. If you
   <emphasis>really</emphasis> want to do it in kernel you should
   regularly check if you need to give up the CPU (remember there is
   cooperative multitasking per CPU).  Idiom:
  </para>

  <programlisting>
cond_resched(); /* Will sleep */
  </programlisting>

  <para>
   A short note on interface design: the UNIX system call motto is
   "Provide mechanism not policy".
  </para>
 </chapter>

 <chapter id="deadlock-recipes">
  <title>Recipes for Deadlock</title>

  <para>
   You cannot call any routines which may sleep, unless:
  </para>
  <itemizedlist>
   <listitem>
    <para>
     You are in user context.
    </para>
   </listitem>

   <listitem>
    <para>
     You do not own any spinlocks.
```

```
    </para>
   </listitem>

   <listitem>
    <para>
     You have interrupts enabled (actually, Andi Kleen says
     that the scheduling code will enable them for you, but
     that's probably not what you wanted).
    </para>
   </listitem>
  </itemizedlist>

  <para>
   Note that some functions may sleep implicitly: common ones are
   the user space access functions (*_user) and memory allocation
   functions without <symbol>GFP_ATOMIC</symbol>.
  </para>

  <para>
   You should always compile your kernel
   <symbol>CONFIG_DEBUG_SPINLOCK_SLEEP</symbol> on, and it will warn
   you if you break these rules.  If you <emphasis>do</emphasis> break
   the rules, you will eventually lock up your box.
  </para>

  <para>
   Really.
  </para>
 </chapter>

 <chapter id="common-routines">
  <title>Common Routines</title>

  <sect1 id="routines-printk">
   <title>
    <function>printk()</function>
    <filename class="headerfile">include/linux/kernel.h</filename>
   </title>

   <para>
    <function>printk()</function> feeds kernel messages to the
    console, dmesg, and the syslog daemon.  It is useful for debugging
    and reporting errors, and can be used inside interrupt context,
    but use with caution: a machine which has its console flooded with
    printk messages is unusable.  It uses a format string mostly
    compatible with ANSI C printf, and C string concatenation to give
    it a first "priority" argument:
   </para>

   <programlisting>
printk(KERN_INFO "i = %u\n", i);
   </programlisting>

   <para>
    See <filename class="headerfile">include/linux/kernel.h</filename>;
    for other KERN_ values; these are interpreted by syslog as the
```

```
    level.  Special case: for printing an IP address use
   </para>

   <programlisting>
__be32 ipaddress;
printk(KERN_INFO "my ip: %pI4\n", &amp;ipaddress);
   </programlisting>

   <para>
    <function>printk()</function> internally uses a 1K buffer and does
    not catch overruns.  Make sure that will be enough.
   </para>

   <note>
    <para>
     You will know when you are a real kernel hacker
     when you start typoing printf as printk in your user programs :)
    </para>
   </note>

   <!--- From the Lions book reader department -->

   <note>
    <para>
     Another sidenote: the original Unix Version 6 sources had a
     comment on top of its printf function: "Printf should not be
     used for chit-chat".  You should follow that advice.
    </para>
   </note>
  </sect1>

  <sect1 id="routines-copy">
   <title>
    <function>copy_[to/from]_user()</function>
    /
    <function>get_user()</function>
    /
    <function>put_user()</function>
    <filename class="headerfile">include/asm/uaccess.h</filename>
   </title>

   <para>
    <emphasis>[SLEEPS]</emphasis>
   </para>

   <para>
    <function>put_user()</function> and <function>get_user()</function>
    are used to get and put single values (such as an int, char, or
    long) from and to userspace.  A pointer into userspace should
    never be simply dereferenced: data should be copied using these
    routines.  Both return <constant>-EFAULT</constant> or 0.
   </para>
   <para>
    <function>copy_to_user()</function> and
    <function>copy_from_user()</function> are more general: they copy
    an arbitrary amount of data to and from userspace.
```

```
    <caution>
     <para>
      Unlike <function>put_user()</function> and
      <function>get_user()</function>, they return the amount of
      uncopied data (ie. <returnvalue>0</returnvalue> still means
      success).
     </para>
    </caution>
    [Yes, this moronic interface makes me cringe.  The flamewar comes up every
year or so. --RR.]
   </para>
   <para>
    The functions may sleep implicitly. This should never be called
    outside user context (it makes no sense), with interrupts
    disabled, or a spinlock held.
   </para>
  </sect1>

  <sect1 id="routines-kmalloc">
   <title><function>kmalloc()</function>/<function>kfree()</function>
    <filename class="headerfile">include/linux/slab.h</filename></title>

   <para>
    <emphasis>[MAY SLEEP: SEE BELOW]</emphasis>
   </para>

   <para>
    These routines are used to dynamically request pointer-aligned
    chunks of memory, like malloc and free do in userspace, but
    <function>kmalloc()</function> takes an extra flag word.
    Important values:
   </para>

   <variablelist>
    <varlistentry>
     <term>
      <constant>
       GFP_KERNEL
      </constant>
     </term>
     <listitem>
      <para>
       May sleep and swap to free memory. Only allowed in user
       context, but is the most reliable way to allocate memory.
      </para>
     </listitem>
    </varlistentry>

    <varlistentry>
     <term>
      <constant>
       GFP_ATOMIC
      </constant>
     </term>
     <listitem>
      <para>
```

```
    Don't sleep. Less reliable than <constant>GFP_KERNEL</constant>,
    but may be called from interrupt context. You should
    <emphasis>really</emphasis> have a good out-of-memory
    error-handling strategy.
   </para>
  </listitem>
 </varlistentry>

 <varlistentry>
  <term>
   <constant>
    GFP_DMA
   </constant>
  </term>
  <listitem>
   <para>
    Allocate ISA DMA lower than 16MB. If you don't know what that
    is you don't need it.  Very unreliable.
   </para>
  </listitem>
 </varlistentry>
</variablelist>

<para>
 If you see a <errorname>sleeping function called from invalid
 context</errorname> warning message, then maybe you called a
 sleeping allocation function from interrupt context without
 <constant>GFP_ATOMIC</constant>.  You should really fix that.
 Run, don't walk.
</para>

<para>
 If you are allocating at least <constant>PAGE_SIZE</constant>
 (<filename class="headerfile">include/asm/page.h</filename>) bytes,
 consider using <function>__get_free_pages()</function>

 (<filename class="headerfile">include/linux/mm.h</filename>).  It
 takes an order argument (0 for page sized, 1 for double page, 2
 for four pages etc.) and the same memory priority flag word as
 above.
</para>

<para>
 If you are allocating more than a page worth of bytes you can use
 <function>vmalloc()</function>.  It'll allocate virtual memory in
 the kernel map.  This block is not contiguous in physical memory,
 but the <acronym>MMU</acronym> makes it look like it is for you
 (so it'll only look contiguous to the CPUs, not to external device
 drivers).  If you really need large physically contiguous memory
 for some weird device, you have a problem: it is poorly supported
 in Linux because after some time memory fragmentation in a running
 kernel makes it hard.  The best way is to allocate the block early
 in the boot process via the <function>alloc_bootmem()</function>
 routine.
</para>
```

```
    <para>
     Before inventing your own cache of often-used objects consider
     using a slab cache in
     <filename class="headerfile">include/linux/slab.h</filename>
    </para>
   </sect1>

   <sect1 id="routines-current">
    <title><function>current</function>
     <filename class="headerfile">include/asm/current.h</filename></title>

    <para>
     This global variable (really a macro) contains a pointer to
     the current task structure, so is only valid in user context.
     For example, when a process makes a system call, this will
     point to the task structure of the calling process.  It is
     <emphasis>not NULL</emphasis> in interrupt context.
    </para>
   </sect1>

   <sect1 id="routines-udelay">
    <title><function>mdelay()</function>/<function>udelay()</function>
      <filename class="headerfile">include/asm/delay.h</filename>
      <filename class="headerfile">include/linux/delay.h</filename>
    </title>

    <para>
     The <function>udelay()</function> and <function>ndelay()</function>
functions can be used for small pauses.
     Do not use large values with them as you risk
     overflow - the helper function <function>mdelay()</function> is useful
     here, or consider <function>msleep()</function>.
    </para>
   </sect1>

   <sect1 id="routines-endian">

<title><function>cpu_to_be32()</function>/<function>be32_to_cpu()</function>/<fu
nction>cpu_to_le32()</function>/<function>le32_to_cpu()</function>
     <filename class="headerfile">include/asm/byteorder.h</filename>
    </title>

    <para>
     The <function>cpu_to_be32()</function> family (where the "32" can
     be replaced by 64 or 16, and the "be" can be replaced by "le") are
     the general way to do endian conversions in the kernel: they
     return the converted value.  All variations supply the reverse as
     well: <function>be32_to_cpu()</function>, etc.
    </para>

    <para>
     There are two major variations of these functions: the pointer
     variation, such as <function>cpu_to_be32p()</function>, which take
     a pointer to the given type, and return the converted value.  The
     other variation is the "in-situ" family, such as
     <function>cpu_to_be32s()</function>, which convert value referred
```

    to by the pointer, and return void.
   </para>
  </sect1>

  <sect1 id="routines-local-irqs">

<title><function>local_irq_save()</function>/<function>local_irq_restore()</function>
    <filename class="headerfile">include/asm/system.h</filename>
   </title>

   <para>
    These routines disable hard interrupts on the local CPU, and
    restore them. They are reentrant; saving the previous state in
    their one <varname>unsigned long flags</varname> argument. If you
    know that interrupts are enabled, you can simply use
    <function>local_irq_disable()</function> and
    <function>local_irq_enable()</function>.
   </para>
  </sect1>

  <sect1 id="routines-softirqs">

<title><function>local_bh_disable()</function>/<function>local_bh_enable()</function>
    <filename class="headerfile">include/linux/interrupt.h</filename></title>

   <para>
    These routines disable soft interrupts on the local CPU, and
    restore them. They are reentrant; if soft interrupts were
    disabled before, they will still be disabled after this pair
    of functions has been called. They prevent softirqs and tasklets
    from running on the current CPU.
   </para>
  </sect1>

  <sect1 id="routines-processorids">
   <title><function>smp_processor_id</function>()
    <filename class="headerfile">include/asm/smp.h</filename></title>

   <para>
    <function>get_cpu()</function> disables preemption (so you won't
    suddenly get moved to another CPU) and returns the current
    processor number, between 0 and <symbol>NR_CPUS</symbol>. Note
    that the CPU numbers are not necessarily continuous. You return
    it again with <function>put_cpu()</function> when you are done.
   </para>
   <para>
    If you know you cannot be preempted by another task (ie. you are
    in interrupt context, or have preemption disabled) you can use
    smp_processor_id().
   </para>
  </sect1>

  <sect1 id="routines-init">
   <title><type>__init</type>/<type>__exit</type>/<type>__initdata</type>

```
  <filename class="headerfile">include/linux/init.h</filename></title>

 <para>
  After boot, the kernel frees up a special section; functions
  marked with <type>__init</type> and data structures marked with
  <type>__initdata</type> are dropped after boot is complete: similarly
  modules discard this memory after initialization.  <type>__exit</type>
  is used to declare a function which is only required on exit: the
  function will be dropped if this file is not compiled as a module.
  See the header file for use. Note that it makes no sense for a function
  marked with <type>__init</type> to be exported to modules with
  <function>EXPORT_SYMBOL()</function> - this will break.
 </para>

</sect1>

<sect1 id="routines-init-again">
 <title><function>__initcall()</function>/<function>module_init()</function>
  <filename class="headerfile">include/linux/init.h</filename></title>
 <para>
  Many parts of the kernel are well served as a module
  (dynamically-loadable parts of the kernel).  Using the
  <function>module_init()</function> and
  <function>module_exit()</function> macros it is easy to write code
  without #ifdefs which can operate both as a module or built into
  the kernel.
 </para>

 <para>
  The <function>module_init()</function> macro defines which
  function is to be called at module insertion time (if the file is
  compiled as a module), or at boot time: if the file is not
  compiled as a module the <function>module_init()</function> macro
  becomes equivalent to <function>__initcall()</function>, which
  through linker magic ensures that the function is called on boot.
 </para>

 <para>
  The function can return a negative error number to cause
  module loading to fail (unfortunately, this has no effect if
  the module is compiled into the kernel).  This function is
  called in user context with interrupts enabled, so it can sleep.
 </para>
</sect1>

<sect1 id="routines-moduleexit">
 <title> <function>module_exit()</function>
  <filename class="headerfile">include/linux/init.h</filename> </title>

 <para>
  This macro defines the function to be called at module removal
  time (or never, in the case of the file compiled into the
  kernel).  It will only be called if the module usage count has
  reached zero.  This function can also sleep, but cannot fail:
  everything must be cleaned up by the time it returns.
 </para>
```

```
    <para>
     Note that this macro is optional: if it is not present, your
     module will not be removable (except for 'rmmod -f').
    </para>
   </sect1>

   <sect1 id="routines-module-use-counters">
    <title>
<function>try_module_get()</function>/<function>module_put()</function>
     <filename class="headerfile">include/linux/module.h</filename></title>

    <para>
     These manipulate the module usage count, to protect against
     removal (a module also can't be removed if another module uses one
     of its exported symbols: see below).  Before calling into module
     code, you should call <function>try_module_get()</function> on
     that module: if it fails, then the module is being removed and you
     should act as if it wasn't there.  Otherwise, you can safely enter
     the module, and call <function>module_put()</function> when you're
     finished.
    </para>

    <para>
     Most registerable structures have an
     <structfield>owner</structfield> field, such as in the
     <structname>file_operations</structname> structure. Set this field
     to the macro <symbol>THIS_MODULE</symbol>.
    </para>
   </sect1>

  <!-- add info on new-style module refcounting here -->
  </chapter>

  <chapter id="queues">
   <title>Wait Queues
    <filename class="headerfile">include/linux/wait.h</filename>
   </title>
   <para>
    <emphasis>[SLEEPS]</emphasis>
   </para>

   <para>
    A wait queue is used to wait for someone to wake you up when a
    certain condition is true.  They must be used carefully to ensure
    there is no race condition.  You declare a
    <type>wait_queue_head_t</type>, and then processes which want to
    wait for that condition declare a <type>wait_queue_t</type>
    referring to themselves, and place that in the queue.
   </para>

   <sect1 id="queue-declaring">
    <title>Declaring</title>

    <para>
     You declare a <type>wait_queue_head_t</type> using the
```

```
   <function>DECLARE_WAIT_QUEUE_HEAD()</function> macro, or using the
   <function>init_waitqueue_head()</function> routine in your
   initialization code.
  </para>
 </sect1>

 <sect1 id="queue-waitqueue">
  <title>Queuing</title>

  <para>
   Placing yourself in the waitqueue is fairly complex, because you
   must put yourself in the queue before checking the condition.
   There is a macro to do this:
   <function>wait_event_interruptible()</function>

   <filename class="headerfile">include/linux/wait.h</filename> The
   first argument is the wait queue head, and the second is an
   expression which is evaluated; the macro returns
   <returnvalue>0</returnvalue> when this expression is true, or
   <returnvalue>-ERESTARTSYS</returnvalue> if a signal is received.
   The <function>wait_event()</function> version ignores signals.
  </para>
  <para>
  Do not use the <function>sleep_on()</function> function family -
  it is very easy to accidentally introduce races; almost certainly
  one of the <function>wait_event()</function> family will do, or a
  loop around <function>schedule_timeout()</function>. If you choose
  to loop around <function>schedule_timeout()</function> remember
  you must set the task state (with
  <function>set_current_state()</function>) on each iteration to avoid
  busy-looping.
  </para>

 </sect1>

 <sect1 id="queue-waking">
  <title>Waking Up Queued Tasks</title>

  <para>
   Call <function>wake_up()</function>

   <filename class="headerfile">include/linux/wait.h</filename>;,
   which will wake up every process in the queue.  The exception is
   if one has <constant>TASK_EXCLUSIVE</constant> set, in which case
   the remainder of the queue will not be woken.  There are other variants
   of this basic function available in the same header.
  </para>
 </sect1>
</chapter>

<chapter id="atomic-ops">
 <title>Atomic Operations</title>

 <para>
  Certain operations are guaranteed atomic on all platforms.  The
  first class of operations work on <type>atomic_t</type>
```

```
  <filename class="headerfile">include/asm/atomic.h</filename>; this
  contains a signed integer (at least 32 bits long), and you must use
  these functions to manipulate or read atomic_t variables.
  <function>atomic_read()</function> and
  <function>atomic_set()</function> get and set the counter,
  <function>atomic_add()</function>,
  <function>atomic_sub()</function>,
  <function>atomic_inc()</function>,
  <function>atomic_dec()</function>, and
  <function>atomic_dec_and_test()</function> (returns
  <returnvalue>true</returnvalue> if it was decremented to zero).
 </para>

 <para>
  Yes.  It returns <returnvalue>true</returnvalue> (i.e. != 0) if the
  atomic variable is zero.
 </para>

 <para>
  Note that these functions are slower than normal arithmetic, and
  so should not be used unnecessarily.
 </para>

 <para>
  The second class of atomic operations is atomic bit operations on an
  <type>unsigned long</type>, defined in

  <filename class="headerfile">include/linux/bitops.h</filename>.  These
  operations generally take a pointer to the bit pattern, and a bit
  number: 0 is the least significant bit.
  <function>set_bit()</function>, <function>clear_bit()</function>
  and <function>change_bit()</function> set, clear, and flip the
  given bit.  <function>test_and_set_bit()</function>,
  <function>test_and_clear_bit()</function> and
  <function>test_and_change_bit()</function> do the same thing,
  except return true if the bit was previously set; these are
  particularly useful for atomically setting flags.
 </para>

 <para>
  It is possible to call these operations with bit indices greater
  than BITS_PER_LONG.  The resulting behavior is strange on big-endian
  platforms though so it is a good idea not to do this.
 </para>
</chapter>

<chapter id="symbols">
 <title>Symbols</title>

 <para>
  Within the kernel proper, the normal linking rules apply
  (ie. unless a symbol is declared to be file scope with the
  <type>static</type> keyword, it can be used anywhere in the
  kernel).  However, for modules, a special exported symbol table is
  kept which limits the entry points to the kernel proper.  Modules
```

```
 can also export symbols.
</para>

<sect1 id="sym-exportsymbols">
 <title><function>EXPORT_SYMBOL()</function>
  <filename class="headerfile">include/linux/module.h</filename></title>

 <para>
  This is the classic method of exporting a symbol: dynamically
  loaded modules will be able to use the symbol as normal.
 </para>
</sect1>

<sect1 id="sym-exportsymbols-gpl">
 <title><function>EXPORT_SYMBOL_GPL()</function>
  <filename class="headerfile">include/linux/module.h</filename></title>

 <para>
  Similar to <function>EXPORT_SYMBOL()</function> except that the
  symbols exported by <function>EXPORT_SYMBOL_GPL()</function> can
  only be seen by modules with a
  <function>MODULE_LICENSE()</function> that specifies a GPL
  compatible license.  It implies that the function is considered
  an internal implementation issue, and not really an interface.
 </para>
</sect1>
</chapter>

<chapter id="conventions">
 <title>Routines and Conventions</title>

 <sect1 id="conventions-doublelinkedlist">
  <title>Double-linked lists
   <filename class="headerfile">include/linux/list.h</filename></title>

  <para>
   There used to be three sets of linked-list routines in the kernel
   headers, but this one is the winner.  If you don't have some
   particular pressing need for a single list, it's a good choice.
  </para>

  <para>
   In particular, <function>list_for_each_entry</function> is useful.
  </para>
 </sect1>

 <sect1 id="convention-returns">
  <title>Return Conventions</title>

  <para>
   For code called in user context, it's very common to defy C
   convention, and return <returnvalue>0</returnvalue> for success,
   and a negative error number
   (eg. <returnvalue>-EFAULT</returnvalue>) for failure.  This can be
   unintuitive at first, but it's fairly widespread in the kernel.
  </para>
```

```
    <para>
     Using <function>ERR_PTR()</function>

     <filename class="headerfile">include/linux/err.h</filename>; to
     encode a negative error number into a pointer, and
     <function>IS_ERR()</function> and <function>PTR_ERR()</function>
     to get it back out again: avoids a separate pointer parameter for
     the error number.  Icky, but in a good way.
    </para>
   </sect1>

   <sect1 id="conventions-borkedcompile">
    <title>Breaking Compilation</title>

    <para>
     Linus and the other developers sometimes change function or
     structure names in development kernels; this is not done just to
     keep everyone on their toes: it reflects a fundamental change
     (eg. can no longer be called with interrupts on, or does extra
     checks, or doesn't do checks which were caught before).  Usually
     this is accompanied by a fairly complete note to the linux-kernel
     mailing list; search the archive.  Simply doing a global replace
     on the file usually makes things <emphasis>worse</emphasis>.
    </para>
   </sect1>

   <sect1 id="conventions-initialising">
    <title>Initializing structure members</title>

    <para>
     The preferred method of initializing structures is to use
     designated initialisers, as defined by ISO C99, eg:
    </para>
    <programlisting>
static struct block_device_operations opt_fops = {
        .open               = opt_open,
        .release            = opt_release,
        .ioctl              = opt_ioctl,
        .check_media_change = opt_media_change,
};
    </programlisting>
    <para>
     This makes it easy to grep for, and makes it clear which
     structure fields are set.  You should do this because it looks
     cool.
    </para>
   </sect1>

   <sect1 id="conventions-gnu-extns">
    <title>GNU Extensions</title>

    <para>
     GNU Extensions are explicitly allowed in the Linux kernel.
     Note that some of the more complex ones are not very well
     supported, due to lack of general use, but the following are
```

```
 considered standard (see the GCC info page section "C
 Extensions" for more details - Yes, really the info page, the
 man page is only a short summary of the stuff in info).
</para>
<itemizedlist>
 <listitem>
  <para>
   Inline functions
  </para>
 </listitem>
 <listitem>
  <para>
   Statement expressions (ie. the ({ and }) constructs).
  </para>
 </listitem>
 <listitem>
  <para>
   Declaring attributes of a function / variable / type
   (__attribute__)
  </para>
 </listitem>
 <listitem>
  <para>
   typeof
  </para>
 </listitem>
 <listitem>
  <para>
   Zero length arrays
  </para>
 </listitem>
 <listitem>
  <para>
   Macro varargs
  </para>
 </listitem>
 <listitem>
  <para>
   Arithmetic on void pointers
  </para>
 </listitem>
 <listitem>
  <para>
   Non-Constant initializers
  </para>
 </listitem>
 <listitem>
  <para>
   Assembler Instructions (not outside arch/ and include/asm/)
  </para>
 </listitem>
 <listitem>
  <para>
   Function names as strings (__func__).
  </para>
 </listitem>
```

```
    <listitem>
     <para>
      __builtin_constant_p()
     </para>
    </listitem>
   </itemizedlist>

   <para>
    Be wary when using long long in the kernel, the code gcc generates for
    it is horrible and worse: division and multiplication does not work
    on i386 because the GCC runtime functions for it are missing from
    the kernel environment.
   </para>

    <!-- FIXME: add a note about ANSI aliasing cleanness -->
  </sect1>

  <sect1 id="conventions-cplusplus">
   <title>C++</title>

   <para>
    Using C++ in the kernel is usually a bad idea, because the
    kernel does not provide the necessary runtime environment
    and the include files are not tested for it.  It is still
    possible, but not recommended.  If you really want to do
    this, forget about exceptions at least.
   </para>
  </sect1>

  <sect1 id="conventions-ifdef">
   <title>&num;if</title>

   <para>
    It is generally considered cleaner to use macros in header files
    (or at the top of .c files) to abstract away functions rather than
    using `#if' pre-processor statements throughout the source code.
   </para>
  </sect1>
 </chapter>

 <chapter id="submitting">
  <title>Putting Your Stuff in the Kernel</title>

  <para>
   In order to get your stuff into shape for official inclusion, or
   even to make a neat patch, there's administrative work to be
   done:
  </para>
  <itemizedlist>
   <listitem>
    <para>
     Figure out whose pond you've been pissing in.  Look at the top of
     the source files, inside the <filename>MAINTAINERS</filename>
     file, and last of all in the <filename>CREDITS</filename> file.
     You should coordinate with this person to make sure you're not
     duplicating effort, or trying something that's already been
```

```
  rejected.
 </para>

 <para>
  Make sure you put your name and EMail address at the top of
  any files you create or mangle significantly.  This is the
  first place people will look when they find a bug, or when
  <emphasis>they</emphasis> want to make a change.
 </para>
</listitem>

<listitem>
 <para>
  Usually you want a configuration option for your kernel hack.
  Edit <filename>Kconfig</filename> in the appropriate directory.
  The Config language is simple to use by cut and paste, and there's
  complete documentation in
  <filename>Documentation/kbuild/kconfig-language.txt</filename>.
 </para>

 <para>
  You may well want to make your CONFIG option only visible if
  <symbol>CONFIG_EXPERIMENTAL</symbol> is enabled: this serves as a
  warning to users.  There many other fancy things you can do: see
  the various <filename>Kconfig</filename> files for ideas.
 </para>

 <para>
  In your description of the option, make sure you address both the
  expert user and the user who knows nothing about your feature.  Mention
  incompatibilities and issues here.  <emphasis> Definitely
  </emphasis> end your description with <quote> if in doubt, say N
  </quote> (or, occasionally, `Y'); this is for people who have no
  idea what you are talking about.
 </para>
</listitem>

<listitem>
 <para>
  Edit the <filename>Makefile</filename>: the CONFIG variables are
  exported here so you can usually just add a "obj-$(CONFIG_xxx) +=
  xxx.o" line.  The syntax is documented in
  <filename>Documentation/kbuild/makefiles.txt</filename>.
 </para>
</listitem>

<listitem>
 <para>
  Put yourself in <filename>CREDITS</filename> if you've done
  something noteworthy, usually beyond a single file (your name
  should be at the top of the source files anyway).
  <filename>MAINTAINERS</filename> means you want to be consulted
  when changes are made to a subsystem, and hear about bugs; it
  implies a more-than-passing commitment to some part of the code.
 </para>
</listitem>
```

```
   <listitem>
    <para>
     Finally, don't forget to read
<filename>Documentation/SubmittingPatches</filename>
     and possibly <filename>Documentation/SubmittingDrivers</filename>.
    </para>
   </listitem>
  </itemizedlist>
 </chapter>

 <chapter id="cantrips">
  <title>Kernel Cantrips</title>

  <para>
   Some favorites from browsing the source.  Feel free to add to this
   list.
  </para>

  <para>
   <filename>arch/x86/include/asm/delay.h:</filename>
  </para>
  <programlisting>
#define ndelay(n) (__builtin_constant_p(n) ? \
        ((n) > 20000 ? __bad_ndelay() : __const_udelay((n) * 5ul)) : \
        __ndelay(n))
  </programlisting>

  <para>
   <filename>include/linux/fs.h</filename>:
  </para>
  <programlisting>
/*
 * Kernel pointers have redundant information, so we can use a
 * scheme where we can return either an error code or a dentry
 * pointer with the same return value.
 *
 * This should be a per-architecture thing, to allow different
 * error and pointer decisions.
 */
 #define ERR_PTR(err)    ((void *)((long)(err)))
 #define PTR_ERR(ptr)    ((long)(ptr))
 #define IS_ERR(ptr)     ((unsigned long)(ptr) > (unsigned long)(-1000))
</programlisting>

  <para>
   <filename>arch/x86/include/asm/uaccess_32.h:</filename>
  </para>

  <programlisting>
#define copy_to_user(to,from,n)                         \
        (__builtin_constant_p(n) ?                      \
         __constant_copy_to_user((to),(from),(n)) :     \
         __generic_copy_to_user((to),(from),(n)))
  </programlisting>
```

```
  <para>
   <filename>arch/sparc/kernel/head.S:</filename>
  </para>

  <programlisting>
/*
 * Sun people can't spell worth damn. "compatability" indeed.
 * At least we *know* we can't spell, and use a spell-checker.
 */

/* Uh, actually Linus it is I who cannot spell. Too much murky
 * Sparc assembly will do this to ya.
 */
C_LABEL(cputypvar):
        .asciz "compatability"

/* Tested on SS-5, SS-10. Probably someone at Sun applied a spell-checker. */
        .align 4
C_LABEL(cputypvar_sun4m):
        .asciz "compatible"
  </programlisting>

  <para>
   <filename>arch/sparc/lib/checksum.S:</filename>
  </para>

  <programlisting>
        /* Sun, you just can't beat me, you just can't.  Stop trying,
         * give up.  I'm serious, I am going to kick the living shit
         * out of you, game over, lights out.
         */
  </programlisting>
 </chapter>

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

  <para>
   Thanks to Andi Kleen for the idea, answering my questions, fixing
   my mistakes, filling content, etc.  Philipp Rumpf for more spelling
   and clarity fixes, and some excellent non-obvious points.  Werner
   Almesberger for giving me a great summary of
   <function>disable_irq()</function>, and Jes Sorensen and Andrea
   Arcangeli added caveats. Michael Elizabeth Chastain for checking
   and adding to the Configure section. <!-- Rusty insisted on this
   bit; I didn't do it! --> Telsa Gwynne for teaching me DocBook.
  </para>
 </chapter>
</book>
```