Unreliable Guide To Hacking The Linux Kernel

Author: Rusty Russell

Introduction

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.

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.

The Players

At any time each of the CPUs in a system can be:

- not associated with any process, serving a hardware interrupt;
- not associated with any process, serving a softirq or tasklet;
- running in kernel space, associated with a process (user context);
- running a process in user space.

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.

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

User Context

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 :c:func:`schedule()`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 50); *backlink*

Unknown interpreted text role "c:func".

Note

You are always in user context on module load and unload, and on operations on the block device layer.

In user context, the `current` pointer (indicating the task we are currently executing) is valid, and :c:func:`in_interrupt()` (`include/linux/preempt.h`) is false.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 59); *backlink*

Unknown interpreted text role "c:func".

Warning

Beware that if you have preemption or softirqs disabled (see below), :c:func:`in_interrupt()` will return a false positive.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 65); *backlink*

Unknown interpreted text role "c:func".

Hardware Interrupts (Hard IRQs)

Timer ticks, network cards and keyboard 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.

You can tell you are in a hardware interrupt, because in_hardirq() returns true.

> **Warning**
>
> Beware that this will return a false positive if interrupts are disabled (see below).

Software Interrupt Context: Softirqs and Tasklets

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 (`kernel/softirq.c`).

Much of the real interrupt handling work is done here. Early in the transition to SMP, 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'.

`include/linux/interrupt.h` lists the different softirqs. A very important softirq is the timer softirq (`include/linux/timer.h`): you can register to have it call functions for you in a given length of time.

Softirqs are often a pain to deal with, since the same softirq will run simultaneously on more than one CPU. For this reason, tasklets (`include/linux/interrupt.h`) 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.

> **Warning**
>
> 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.

You can tell you are in a softirq (or tasklet) using the :c:func:`in_softirq()` macro (`include/linux/preempt.h`).

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking) hacking.rst`, line 118); *backlink***
>
> Unknown interpreted text role "c:func".

> **Warning**
>
> Beware that this will return a false positive if a :ref:`botton half lock <local_bh_disable>` is held.
>
> > **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking) hacking.rst`, line 123); *backlink***
> >
> > Unknown interpreted text role "ref".

Some Basic Rules

No memory protection
: 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?

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

A rigid stack limit
: 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).

The Linux kernel is portable

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.

ioctls: Not writing a new system call

A system call generally looks like this:

```
asmlinkage long sys_mycall(int arg)
{
        return 0;
}
```

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 `include/asm/unistd.h` and `arch/kernel/entry.S` file, and is much more likely to be accepted by Linus.

If all your routine does is read or write some parameter, consider implementing a :c:func:`sysfs()` interface instead.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 174); *backlink*

Unknown interpreted text role "c:func".

Inside the ioctl you're in user context to a process. When a error occurs you return a negated errno (see `include/uapi/asm-generic/errno-base.h`, `include/uapi/asm-generic/errno.h` and `include/linux/errno.h`), otherwise you return 0.

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 `-ERESTARTSYS` 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.

```
if (signal_pending(current))
        return -ERESTARTSYS;
```

If you're doing longer computations: first think userspace. If you **really** 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:

```
cond_resched(); /* Will sleep */
```

A short note on interface design: the UNIX system call motto is "Provide mechanism not policy".

Recipes for Deadlock

You cannot call any routines which may sleep, unless:

- You are in user context.
- You do not own any spinlocks.
- 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).

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

You should always compile your kernel `CONFIG_DEBUG_ATOMIC_SLEEP` on, and it will warn you if you break these rules. If you **do** break the rules, you will eventually lock up your box.

Really.

Common Routines

:c:func:`printk()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 234); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/printk.h`

:c:func:`printk()` 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:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 239); *backlink*

Unknown interpreted text role "c:func".

```
printk(KERN_INFO "i = %u\n", i);
```

See `include/linux/kern_levels.h`; for other `KERN_` values; these are interpreted by syslog as the level. Special case: for printing an IP address use:

```
__be32 ipaddress;
printk(KERN_INFO "my ip: %pI4\n", &ipaddress);
```

:c:func:`printk()` internally uses a 1K buffer and does not catch overruns. Make sure that will be enough.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 257); *backlink*

Unknown interpreted text role "c:func".

Note

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

Note

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.

:c:func:`copy_to_user()` / :c:func:`copy_from_user()` / :c:func:`get_user()` / :c:func:`put_user()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 271); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 271); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 271); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 271); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/uaccess.h` / `asm/uaccess.h`

[SLEEPS]

:c:func:`put_user()` and :c:func:`get_user()` 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 `-EFAULT` or 0.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 278); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 278); *backlink*

Unknown interpreted text role "c:func".

:c:func:`copy_to_user()` and :c:func:`copy_from_user()` are more general: they copy an arbitrary amount of data to and from userspace.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 284); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 284); *backlink*

Unknown interpreted text role "c:func".

Warning

Unlike :c:func:`put_user()` and :c:func:`get_user()`, they return the amount of uncopied data (ie. 0 still means success).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 290); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 290); *backlink*

Unknown interpreted text role "c:func".

[Yes, this moronic interface makes me cringe. The flamewar comes up every year or so. --RR.]

The functions may sleep implicitly. This should never be called outside user context (it makes no sense), with interrupts disabled, or a spinlock held.

:c:func:`kmalloc()`/:c:func:`kfree()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 300); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 300); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/slab.h`

[MAY SLEEP: SEE BELOW]

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

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 307); *backlink*
>
> Unknown interpreted text role "c:func".

`GFP_KERNEL`
: May sleep and swap to free memory. Only allowed in user context, but is the most reliable way to allocate memory.

`GFP_ATOMIC`
: Don't sleep. Less reliable than `GFP_KERNEL`, but may be called from interrupt context. You should **really** have a good out-of-memory error-handling strategy.

`GFP_DMA`
: Allocate ISA DMA lower than 16MB. If you don't know what that is you don't need it. Very unreliable.

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

If you are allocating at least `PAGE_SIZE` (`asm/page.h` or `asm/page_types.h`) bytes, consider using :c:func:`__get_free_pages()` (`include/linux/gfp.h`). 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.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 329); *backlink*
>
> Unknown interpreted text role "c:func".

If you are allocating more than a page worth of bytes you can use :c:func:`vmalloc()`. It'll allocate virtual memory in the kernel map. This block is not contiguous in physical memory, but the MMU 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 :c:func:`alloc_bootmem()` routine.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 335); *backlink*
>
> Unknown interpreted text role "c:func".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 335); *backlink*
>
> Unknown interpreted text role "c:func".

Before inventing your own cache of often-used objects consider using a slab cache in `include/linux/slab.h`

:c:macro:`current`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 349); *backlink*
>
> Unknown interpreted text role "c:macro".

Defined in `include/asm/current.h`

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 **not NULL** in interrupt context.

:c:func:`mdelay()`/:c:func:`udelay()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 359); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 359); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/asm/delay.h` / `include/linux/delay.h`

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 364); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 364); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 364); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 364); *backlink*

Unknown interpreted text role "c:func".

:c:func:`cpu_to_be32()`/:c:func:`be32_to_cpu()`/:c:func:`cpu_to_le32()`/:c:func:`le32_to_cpu()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 369); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 369); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 369); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-`

master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 369); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/asm/byteorder.h`

The :c:func:`cpu_to_be32()` 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: :c:func:`be32_to_cpu()`, etc.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 374); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 374); *backlink*

Unknown interpreted text role "c:func".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 380); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 380); *backlink*

Unknown interpreted text role "c:func".

:c:func:`local_irq_save()`/:c:func:`local_irq_restore()`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 386); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 386); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/irqflags.h`

These routines disable hard interrupts on the local CPU, and restore them. They are reentrant; saving the previous state in their one `unsigned long flags` argument. If you know that interrupts are enabled, you can simply use :c:func:`local_irq_disable()` and :c:func:`local_irq_enable()`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 391); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst, line 391); *backlink*

Unknown interpreted text role "c:func".

:c:func:`local_bh_disable()`/:c:func:`local_bh_enable()`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst, line 399); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst, line 399); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/bottom_half.h`

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.

:c:func:`smp_processor_id()`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst, line 410); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/smp.h`

:c:func:`get_cpu()` disables preemption (so you won't suddenly get moved to another CPU) and returns the current processor number, between 0 and `NR_CPUS`. Note that the CPU numbers are not necessarily continuous. You return it again with :c:func:`put_cpu()` when you are done.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst, line 415); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst, line 415); *backlink*

Unknown interpreted text role "c:func".

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().

`__init`/`__exit`/`__initdata`

Defined in `include/linux/init.h`

After boot, the kernel frees up a special section; functions marked with `__init` and data structures marked with `__initdata` are dropped after boot is complete: similarly modules discard this memory after initialization. `__exit` 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 `__init` to be exported to modules with :c:func:`EXPORT_SYMBOL()` or :c:func:`EXPORT_SYMBOL_GPL()`- this will break.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst, line 430); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 430); *backlink*

Unknown interpreted text role "c:func".

:c:func:`__initcall()`/:c:func:`module_init()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 440); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 440); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/init.h` / `include/linux/module.h`

Many parts of the kernel are well served as a module (dynamically-loadable parts of the kernel). Using the :c:func:`module_init()` and :c:func:`module_exit()` macros it is easy to write code without #ifdefs which can operate both as a module or built into the kernel.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 445); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 445); *backlink*

Unknown interpreted text role "c:func".

The :c:func:`module_init()` 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 :c:func:`module_init()` macro becomes equivalent to :c:func:`__initcall()`, which through linker magic ensures that the function is called on boot.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 451); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 451); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 451); *backlink*

Unknown interpreted text role "c:func".

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.

:c:func:`module_exit()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 463); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/module.h`

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.

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

:c:func:`try_module_get()`/:c:func:`module_put()`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 478); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 478); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/module.h`

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 :c:func:`try_module_get()` 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 :c:func:`module_put()` when you're finished.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 483); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 483); *backlink*

Unknown interpreted text role "c:func".

Most registerable structures have an owner field, such as in the :c:type:`struct file_operations <file_operations>` structure. Set this field to the macro `THIS_MODULE`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 491); *backlink*

Unknown interpreted text role "c:type".

Wait Queues `include/linux/wait.h`

[SLEEPS]

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 :c:type:`wait_queue_head_t`, and then processes which want to wait for that condition declare a :c:type:`wait_queue_entry_t` referring to themselves, and place that in the queue.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 500); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 500); *backlink*

Unknown interpreted text role "c:type".

Declaring

You declare a `wait_queue_head_t` using the :c:func:`DECLARE_WAIT_QUEUE_HEAD()` macro, or using the :c:func:`init_waitqueue_head()` routine in your initialization code.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 509); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 509); *backlink*

Unknown interpreted text role "c:func".

Queuing

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: :c:func:`wait_event_interruptible()` (`include/linux/wait.h`) The first argument is the wait queue head, and the second is an expression which is evaluated; the macro returns 0 when this expression is true, or `-ERESTARTSYS` if a signal is received. The :c:func:`wait_event()` version ignores signals.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 517); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 517); *backlink*

Unknown interpreted text role "c:func".

Waking Up Queued Tasks

Call :c:func:`wake_up()` (`include/linux/wait.h`), which will wake up every process in the queue. The exception is if one has `TASK_EXCLUSIVE` 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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 528); *backlink*

Unknown interpreted text role "c:func".

Atomic Operations

Certain operations are guaranteed atomic on all platforms. The first class of operations work on :c:type:`atomic_t` (`include/asm/atomic.h`); this contains a signed integer (at least 32 bits long), and you must use these functions to manipulate or read :c:type:`atomic_t` variables. :c:func:`atomic_read()` and :c:func:`atomic_set()` get and set the counter, :c:func:`atomic_add()`, :c:func:`atomic_sub()`, :c:func:`atomic_inc()`, :c:func:`atomic_dec()`, and :c:func:`atomic_dec_and_test()` (returns true if it was decremented to zero).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking) hacking.rst, line 537); *backlink*

Unknown interpreted text role "c:func".

Yes. It returns true (i.e. != 0) if the atomic variable is zero.

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

The second class of atomic operations is atomic bit operations on an `unsigned long`, defined in `include/linux/bitops.h`. These operations generally take a pointer to the bit pattern, and a bit number: 0 is the least significant bit. :c:func:`set_bit()`, :c:func:`clear_bit()` and :c:func:`change_bit()` set, clear, and flip the given bit. :c:func:`test_and_set_bit()`, :c:func:`test_and_clear_bit()` and :c:func:`test_and_change_bit()` do the same thing, except return true if the bit was previously set; these are particularly useful for atomically setting flags.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\ (linux-master) (Documentation) (kernel-hacking)hacking.rst, line 552); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst, line 552); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst, line 552); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst, line 552); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst, line 552); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst, line 552); *backlink*

Unknown interpreted text role "c:func".

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.

Symbols

Within the kernel proper, the normal linking rules apply (ie. unless a symbol is declared to be file scope with the `static` 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.

:c:func:`EXPORT_SYMBOL()`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst, line 576); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/export.h`

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

:c:func:`EXPORT_SYMBOL_GPL()`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst, line 584); *backlink*

Unknown interpreted text role "c:func".

Defined in `include/linux/export.h`

Similar to :c:func:`EXPORT_SYMBOL()` except that the symbols exported by :c:func:`EXPORT_SYMBOL_GPL()` can only be seen by modules with a :c:func:`MODULE_LICENSE()` that specifies a GPL compatible license. It implies that the function is considered an internal implementation issue, and not really an interface. Some maintainers and developers may however require EXPORT_SYMBOL_GPL() when adding any new APIs or functionality.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 589); *backlink*
>
> Unknown interpreted text role "c:func".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 589); *backlink*
>
> Unknown interpreted text role "c:func".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 589); *backlink*
>
> Unknown interpreted text role "c:func".

:c:func:`EXPORT_SYMBOL_NS()`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 597); *backlink*
>
> Unknown interpreted text role "c:func".

Defined in `include/linux/export.h`

This is the variant of *EXPORT_SYMBOL()* that allows specifying a symbol namespace. Symbol Namespaces are documented in Documentation/core-api/symbol-namespaces.rst

:c:func:`EXPORT_SYMBOL_NS_GPL()`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 606); *backlink*
>
> Unknown interpreted text role "c:func".

Defined in `include/linux/export.h`

This is the variant of *EXPORT_SYMBOL_GPL()* that allows specifying a symbol namespace. Symbol Namespaces are documented in Documentation/core-api/symbol-namespaces.rst

Routines and Conventions

Double-linked lists `include/linux/list.h`

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.

In particular, :c:func:`list_for_each_entry()` is useful.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 625); *backlink*
>
> Unknown interpreted text role "c:func".

Return Conventions

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

Using :c:func:`ERR_PTR()` (`include/linux/err.h`) to encode a negative error number into a pointer, and :c:func:`IS_ERR()` and :c:func:`PTR_ERR()` to get it back out again: avoids a separate pointer parameter for the error number. Icky, but in a good way.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 635); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 635); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master) (Documentation) (kernel-hacking)hacking.rst`, line 635); *backlink*

Unknown interpreted text role "c:func".

Breaking Compilation

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 **worse**.

Initializing structure members

The preferred method of initializing structures is to use designated initialisers, as defined by ISO C99, eg:

```
static struct block_device_operations opt_fops = {
        .open               = opt_open,
        .release            = opt_release,
        .ioctl              = opt_ioctl,
        .check_media_change = opt_media_change,
};
```

This makes it easy to grep for, and makes it clear which structure fields are set. You should do this because it looks cool.

GNU Extensions

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).

- Inline functions
- Statement expressions (ie. the ({ and }) constructs).
- Declaring attributes of a function / variable / type (__attribute__)
- typeof
- Zero length arrays
- Macro varargs
- Arithmetic on void pointers
- Non-Constant initializers
- Assembler Instructions (not outside arch/ and include/asm/)
- Function names as strings (__func__).
- __builtin_constant_p()

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.

C++

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.

#if

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.

Putting Your Stuff in the Kernel

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:

- Figure out whose pond you've been pissing in. Look at the top of the source files, inside the `MAINTAINERS` file, and last of all in the `CREDITS` file. You should coordinate with this person to make sure you're not duplicating effort, or trying something that's already been rejected.

  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 **they** want to make a change.

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

  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. **Definitely** end your description with "if in doubt, say N" (or, occasionally, `Y'); this is for people who have no idea what you are talking about.

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

- Put yourself in `CREDITS` if you've done something noteworthy, usually beyond a single file (your name should be at the top of the source files anyway). `MAINTAINERS` 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.

- Finally, don't forget to read `Documentation/process/submitting-patches.rst` and possibly `Documentation/process/submitting-drivers.rst`.

Kernel Cantrips

Some favorites from browsing the source. Feel free to add to this list.

`arch/x86/include/asm/delay.h`:

```
#define ndelay(n) (__builtin_constant_p(n) ? \
        ((n) > 20000 ? __bad_ndelay() : __const_udelay((n) * 5ul)) : \
        __ndelay(n))
```

`include/linux/fs.h`:

```
/*
 * 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))
```

`arch/x86/include/asm/uaccess_32.h:`:

```
#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)))
```

`arch/sparc/kernel/head.S:`:

```
/*
 * 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 "compatibility"

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

```
arch/sparc/lib/checksum.S::

   /* 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.
    */
```

Thanks

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 :c:func:`disable_irq()`, and Jes Sorensen and Andrea Arcangeli added caveats. Michael Elizabeth Chastain for checking and adding to the Configure section. Telsa Gwynne for teaching me DocBook.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\kernel-hacking\(linux-master)(Documentation)(kernel-hacking)hacking.rst`, line 825); ***backlink***

Unknown interpreted text role "c:func".