Unreliable Guide To Hacking The Linux Kernel

Paul Rusty Russell

rusty@rustcorp.com.au

Unreliable Guide To Hacking The Linux Kernel
by Paul Rusty Russell

Table of Contents

Chapter 1. Introduction

Welcome, gentle reader, to Rusty's 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.

Chapter 2. 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, tasklet or bh;
- running in kernel space, associated with a process;
- running a process in user space.

There is a strict ordering between these: other than the last category (userspace) each can only be pre-empted by those above. For example, while a softirq is running on a CPU, no other softirq will pre-empt 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: you can sleep, and you own the CPU (except for interrupts) until you call `schedule()`. In other words, user context (unlike userspace) is not pre-emptable.

> **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 `in_interrupt()` (`include/asm/hardirq.h`) is false .

> **Caution**
>
> Beware that if you have interrupts or bottom halves disabled (see below), `in_interrupt()` will return a false positive.

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 another 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_irq()` returns true.

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

Software Interrupt Context: Bottom Halves, Tasklets, softirqs

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.

`include/linux/interrupt.h` lists the different BH's. No matter how many CPUs you have, no two BHs will run at the same time. This made the transition to SMP simpler, but sucks hard for scalable performance. A very important bottom half is the timer BH (`include/linux/timer.h`): you can register to have it call functions for you in a given length of time.

2.3.43 introduced softirqs, and re-implemented the (now deprecated) BHs underneath them. Softirqs are fully-SMP versions of BHs: they can run on as many CPUs at once as required. This means they need to deal with any races in shared data using their own locks. A bitmask is used to keep track of which are enabled, so the 32 available softirqs should not be used up lightly. (*Yes*, people will notice).

tasklets (`include/linux/interrupt.h`) are like softirqs, except 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 (unlike different BHs).

> **Caution**
>
> 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 bottom half, or tasklet) using the `in_softirq()` macro (`include/asm/softirq.h`).

> **Caution**
>
> Beware that this will return a false positive if a bh lock (see below) is held.

Chapter 3. 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

The kernel stack is about 6K in 2.2 (for most architectures: it's about 14K on the Alpha), and 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.

Chapter 4. ioctls: Not writing a new system call

A system call generally looks like this

```
asmlinkage int 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 `sysctl` interface instead.

Inside the ioctl you're in user context to a process. When a error occurs you return a negated errno (see `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())
        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:

```
if (current->need_resched)
        schedule(); /* Will sleep */
```

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

Chapter 5. 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 will eventually lock up your box if you break these rules.

Really.

Chapter 6. Common Routines

printk() include/linux/kernel.h

`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:

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

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

```
__u32 ipaddress;
printk(KERN_INFO "my ip: %d.%d.%d.%d\n", NIPQUAD(ipaddress));
```

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

> **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.

copy_[to/from]_user() / get_user() / put_user() include/asm/uaccess.h

[SLEEPS]

`put_user()` and `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.

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

> **Caution**
>
> Unlike `put_user()` and `get_user()`, they return the amount of uncopied data (ie. 0 still means success).

[Yes, this moronic interface makes me cringe. Please submit a patch and become my hero –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.

kmalloc()/kfree() 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 `kmalloc()` takes an extra flag word. Important values:

`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 kmem_grow: Called nonatomically from int warning message you called a memory 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` (`include/asm/page.h`) bytes, consider using `__get_free_pages()` (`include/linux/mm.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.

If you are allocating more than a page worth of bytes you can use `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 `alloc_bootmem()` routine.

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

current 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.

udelay()/mdelay() include/asm/delay.h include/linux/delay.h

The `udelay()` function can be used for small pauses. Do not use large values with `udelay()` as you risk overflow - the helper function `mdelay()` is useful here, or even consider `schedule_timeout()`.

`local_irq_save()`/`local_irq_restore()` `include/asm/system.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 `local_irq_disable()` and `local_irq_enable()`.

`local_bh_disable()`/`local_bh_enable()` `include/asm/softirq.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, tasklets and bottom halves from running on the current CPU.

`smp_processor_id()`/`cpu_[number/logical]_map()` `include/asm/smp.h`

`smp_processor_id()` returns the current processor number, between 0 and NR_CPUS (the maximum number of CPUs supported by Linux, currently 32). These values are not necessarily continuous: to get a number between 0 and `smp_num_cpus()` (the number of actual processors in this machine), the `cpu_number_map()` function is used to map the processor id to a logical number. `cpu_logical_map()` does the reverse.

__init/__exit/__initdata `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 (within modules this directive is currently ignored). __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 `EXPORT_SYMBOL()` - this will break.

Static data structures marked as __initdata must be initialised (as opposed to ordinary static data which is zeroed BSS) and cannot be const.

`__initcall()`/`module_init()` `include/linux/init.h`

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

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

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). For modules, this is called in user context, with interrupts enabled, and the kernel lock held, so it can sleep.

module_exit() include/linux/init.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.

MOD_INC_USE_COUNT/MOD_DEC_USE_COUNT 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). Every reference to the module from user context should be reflected by this counter (e.g. for every data structure or socket) before the function sleeps. To quote Tim Waugh:

```
/* THIS IS BAD */
foo_open (...)
{
        stuff..
        if (fail)
                return -EBUSY;
        sleep.. (might get unloaded here)
        stuff..
        MOD_INC_USE_COUNT;
        return 0;
}

/* THIS IS GOOD /
foo_open (...)
{
        MOD_INC_USE_COUNT;
        stuff..
        if (fail) {
                MOD_DEC_USE_COUNT;
                return -EBUSY;
        }
        sleep.. (safe now)
        stuff..
        return 0;
}
```

You can often avoid having to deal with these problems by using the *owner* field of the file_operations structure. Set this field as the macro THIS_MODULE.

For more complicated module unload locking requirements, you can set the *can_unload* function pointer to your own routine, which should return 0 if the module is unloadable, or -EBUSY otherwise.

Chapter 7. 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 wait_queue_head_t, and then processes which want to wait for that condition declare a wait_queue_t referring to themselves, and place that in the queue.

Declaring

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

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: `wait_event_interruptible()` `include/linux/sched.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 `wait_event()` version ignores signals.

Do not use the `sleep_on()` function family - it is very easy to accidentally introduce races; almost certainly one of the `wait_event()` family will do, or a loop around `schedule_timeout()`. If you choose to loop around `schedule_timeout()` remember you must set the task state (with `set_current_state()`) on each iteration to avoid busy-looping.

Waking Up Queued Tasks

Call `wake_up()` `include/linux/sched.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.

Chapter 8. Atomic Operations

Certain operations are guaranteed atomic on all platforms. The first class of operations work on atomic_t `include/asm/atomic.h`; this contains a signed integer (at least 24 bits long), and you must use these functions to manipulate or read atomic_t variables. `atomic_read()` and `atomic_set()` get and set the counter, `atomic_add()`, `atomic_sub()`,`atomic_inc()`,`atomic_dec()`,and `atomic_dec_and_test()` (returns true if it was decremented to zero).

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. On some platforms they are much slower, like 32-bit Sparc where they use a spinlock.

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

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.

Note that the order of bits depends on the architecture, and in particular, the bitfield passed to these operations must be at least as large as a long.

Chapter 9. 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.

`EXPORT_SYMBOL()` `include/linux/module.h`

This is the classic method of exporting a symbol, and it works for both modules and non-modules. In the kernel all these declarations are often bundled into a single file to help genksyms (which searches source files for these declarations). See the comment on genksyms and Makefiles below.

EXPORT_NO_SYMBOLS `include/linux/module.h`

If a module exports no symbols then you can specify

```
EXPORT_NO_SYMBOLS;
```

anywhere in the module. In kernel 2.4 and earlier, if a module contains neither `EXPORT_SYMBOL()` nor EXPORT_NO_SYMBOLS then the module defaults to exporting all non-static global symbols. In kernel 2.5 onwards you must explicitly specify whether a module exports symbols or not.

`EXPORT_SYMBOL_GPL()` `include/linux/module.h`

Similar to `EXPORT_SYMBOL()` except that the symbols exported by `EXPORT_SYMBOL_GPL()` can only be seen by modules with a `MODULE_LICENCE()` that specifies a GPL compatible license.

Chapter 10. Routines and Conventions

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

There are three sets of linked-list routines in the kernel headers, but this one seems to be winning out (and Linus has used it). If you don't have some particular pressing need for a single list, it's a good choice. In fact, I don't care whether it's a good choice or not, just use it so we can get rid of the others.

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 networking code, for example.

The filesystem code uses `ERR_PTR()` `include/linux/fs.h`; to encode a negative error number into a pointer, and `IS_ERR()` and `PTR_ERR()` to get it back out again: avoids a separate pointer parameter for the error number. Icky, but in a good way.

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 the gcc Labeled Elements extension, 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__)
- Labeled elements
- 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 (__FUNCTION__)
- __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.

Chapter 11. 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 `Config.in` in the appropriate directory (but under `arch/` it's called `config.in`). The Config Language used is not bash, even though it looks like bash; the safe way is to use only the constructs that you already see in `Config.in` files (see `Documentation/kbuild/config-language.txt`). It's good to run "make xconfig" at least once to test (because it's the only one with a static parser).

  Variables which can be Y or N use bool followed by a tagline and the config define name (which must start with CONFIG_). The tristate function is the same, but allows the answer M (which defines CONFIG_foo_MODULE in your source, instead of CONFIG_FOO) if CONFIG_MODULES is enabled.

  You may well want to make your CONFIG option only visible if CONFIG_EXPERIMENTAL is enabled: this serves as a warning to users. There many other fancy things you can do: see the various `Config.in` files for ideas.

- Edit the `Makefile`: the CONFIG variables are exported here so you can conditionalize compilation with 'ifeq'. If your file exports symbols then add the names to `export-objs` so that genksyms will find them.

  > **Caution**
  >
  > There is a restriction on the kernel build system that objects which export symbols must have globally unique names. If your object does not have a globally unique name then the standard fix is to move the `EXPORT_SYMBOL()` statements to their own object with a unique name. This is why several systems have separate exporting objects, usually suffixed with ksyms.

- Document your option in Documentation/Configure.help. 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.
- 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/SubmittingPatches` and possibly `Documentation/SubmittingDrivers`.

Chapter 12. Kernel Cantrips

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

```
include/linux/brlock.h:

extern inline void br_read_lock (enum brlock_indices idx)
{
        /*
         * This causes a link-time bug message if an
         * invalid index is used:
         */
        if (idx >= __BR_END)
                __br_lock_usage_bug();

        read_lock(&__brlock_array[smp_processor_id()][idx]);
}
```

```
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))
```

```
include/asm-i386/uaccess.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 "compatability"

/* 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.
 */
```

Chapter 13. 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 `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.