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.

2.1. 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.

2.2. 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).

2.3. 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

6.1. `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.

6.2. `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.

6.3. `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`