Unreliable Guide To Hacking The Linux Kernel

Release

The kernel development community

August 18, 2018

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 underqualified, 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 `schedule()`.

> **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/linux/preempt.h) is false.

Warning:	*Beware that if you have preemption or softirqs disabled (see below), `in_interrupt()` will ret* *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 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_irq()` 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 t* *with some bad vodka Alexey Kuznetsov had at the time.*

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

Warning:	*Beware that this will return a false positive if a botton half lock is held.*

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 `sysfs()` interface instead.

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

printk()

Defined in include/linux/printk.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/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);
```

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_user() / copy_from_user() / get_user() / put_user()

Defined in include/linux/uaccess.h / 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.

> **Warning:**   Unlike put_user() and get_user(), they return the amount of uncopied data (ie. 0 still m
>   success).

[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.

kmalloc()/kfree()

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 `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 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 `__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.

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()

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.

mdelay()/udelay()

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

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

cpu_to_be32()/be32_to_cpu()/cpu_to_le32()/le32_to_cpu()

Defined in `include/asm/byteorder.h`

The `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: `be32_to_cpu()`, etc.

There are two major variations of these functions: the pointer variation, such as `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 `cpu_to_be32s()`, which convert value referred to by the pointer, and return void.

local_irq_save()/local_irq_restore()

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 `local_irq_disable()` and `local_irq_enable()`.

local_bh_disable()/local_bh_enable()

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.

smp_processor_id()

Defined in `include/linux/smp.h`

`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 `put_cpu()` when you are done.

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 EXPORT_SYMBOL() or EXPORT_SYMBOL_GPL()- this will break.

__initcall()/module_init()

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 `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). This function is called in user context with interrupts enabled, so it can sleep.

module_exit()

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').

try_module_get()/module_put()

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 `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 `module_put()` when you're finished.

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

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_entry_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/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 `wait_event()` version ignores signals.

Waking Up Queued Tasks

Call 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.

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 32 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.

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. `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 atomically setting flags.

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.

EXPORT_SYMBOL()

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.

EXPORT_SYMBOL_GPL()

Defined in `include/linux/export.h`

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

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, `list_for_each_entry()` is useful.

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 ERR_PTR() (include/linux/err.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 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.txt.

  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.txt.

- 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 `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.

UNRELIABLE GUIDE TO LOCKING

Author Rusty Russell

Introduction

Welcome, to Rusty's Remarkably Unreliable Guide to Kernel Locking issues. This document describes the locking systems in the Linux Kernel in 2.6.

With the wide availability of HyperThreading, and preemption in the Linux Kernel, everyone hacking on the kernel needs to know the fundamentals of concurrency and locking for SMP.

The Problem With Concurrency

(Skip this if you know what a Race Condition is).

In a normal program, you can increment a counter like so:

```
very_important_count++;
```

This is what they would expect to happen:

Table 2.1: Expected Results

Instance 1	Instance 2
read very_important_count (5)	
add 1 (6)	
write very_important_count (6)	
	read very_important_count (6)
	add 1 (7)
	write very_important_count (7)

This is what might happen:

Table 2.2: Possible Results

Instance 1	Instance 2
read very_important_count (5)	
	read very_important_count (5)
add 1 (6)	
	add 1 (6)
write very_important_count (6)	
	write very_important_count (6)

Race Conditions and Critical Regions

This overlap, where the result depends on the relative timing of multiple tasks, is called a race condition. The piece of code containing the concurrency issue is called a critical region. And especially since Linux starting running on SMP machines, they became one of the major issues in kernel design and implementation.

Preemption can have the same effect, even if there is only one CPU: by preempting one task during the critical region, we have exactly the same race condition. In this case the thread which preempts might run the critical region itself.

The solution is to recognize when these simultaneous accesses occur, and use locks to make sure that only one instance can enter the critical region at any time. There are many friendly primitives in the Linux kernel to help you do this. And then there are the unfriendly primitives, but I'll pretend they don't exist.

Locking in the Linux Kernel

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

Be reluctant to introduce new locks.

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

Two Main Types of Kernel Locks: Spinlocks and Mutexes

There are two main types of kernel locks. The fundamental type is the spinlock (`include/asm/spinlock.h`), which is a very simple single-holder lock: if you can't get the spinlock, you keep trying (spinning) until you can. Spinlocks are very small and fast, and can be used anywhere.

The second type is a mutex (`include/linux/mutex.h`): it is like a spinlock, but you may block holding a mutex. If you can't lock a mutex, your task will suspend itself, and be woken up when the mutex is released. This means the CPU can do something else while you are waiting. There are many cases when you simply can't sleep (see *What Functions Are Safe To Call From Interrupts?*), and so have to use a spinlock instead.

Neither type of lock is recursive: see *Deadlock: Simple and Advanced*.

Locks and Uniprocessor Kernels

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

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

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

Mutexes still exist, because they are required for synchronization between user contexts, as we will see below.

Locking Only In User Context

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

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

Locking Between User Context and Softirqs

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

Note that you can also use spin_lock_irq() or spin_lock_irqsave() here, which stop hardware interrupts as well: see *Hard IRQ Context*.

This works perfectly for UP as well: the spin lock vanishes, and this macro simply becomes local_bh_disable() (include/linux/interrupt.h), which protects you from the softirq being run.

Locking Between User Context and Tasklets

This is exactly the same as above, because tasklets are actually run from a softirq.

Locking Between User Context and Timers

This, too, is exactly the same as above, because timers are actually run from a softirq. From a locking point of view, tasklets and timers are identical.

Locking Between Tasklets/Timers

Sometimes a tasklet or timer might want to share data with another tasklet or timer.

The Same Tasklet/Timer

Since a tasklet is never run on two CPUs at once, you don't need to worry about your tasklet being reentrant (running twice at once), even on SMP.

Different Tasklets/Timers

If another tasklet/timer wants to share data with your tasklet or timer , you will both need to use spin_lock() and spin_unlock() calls. spin_lock_bh() is unnecessary here, as you are already in a tasklet, and none will be run on the same CPU.

Locking Between Softirqs

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

The Same Softirq

The same softirq can run on the other CPUs: you can use a per-CPU array (see *Per-CPU Data*) for better performance. If you're going so far as to use a softirq, you probably care about scalable performance enough to justify the extra complexity.

You'll need to use `spin_lock()` and `spin_unlock()` for shared data.

Different Softirqs

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

Hard IRQ Context

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

Locking Between Hard IRQ and Softirqs/Tasklets

If a hardware irq handler shares data with a softirq, you have two concerns. Firstly, the softirq processing can be interrupted by a hardware interrupt, and secondly, the critical region could be entered by a hardware interrupt on another CPU. This is where `spin_lock_irq()` is used. It is defined to disable interrupts on that cpu, then grab the lock. `spin_unlock_irq()` does the reverse.

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

This works perfectly for UP as well: the spin lock vanishes, and this macro simply becomes `local_irq_disable()` (`include/asm/smp.h`), which protects you from the softirq/tasklet/BH being run.

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

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

Locking Between Two Hard IRQ Handlers

It is rare to have to share data between two IRQ handlers, but if you do, `spin_lock_irqsave()` should be used: it is architecture-specific whether all interrupts are disabled inside irq handlers themselves.

Cheat Sheet For Locking

Pete Zaitcev gives the following summary:

- If you are in a process context (any syscall) and want to lock other process out, use a mutex. You can take a mutex and sleep (`copy_from_user*(` or `kmalloc(x,GFP_KERNEL)`).
- Otherwise (== data can be touched in an interrupt), use `spin_lock_irqsave()` and `spin_unlock_irqrestore()`.

- Avoid holding spinlock for more than 5 lines of code and across any function call (except accessors like readb()).

Table of Minimum Requirements

The following table lists the **minimum** locking requirements between various contexts. In some cases, the same context can only be running on one CPU at a time, so no locking is required for that context (eg. a particular thread can only run on one CPU at a time, but if it needs shares data with another thread, locking is required).

Remember the advice above: you can always use spin_lock_irqsave(), which is a superset of all other spinlock primitives.

.	IRQ Handler A	IRQ Handler B	Softirq A	Softirq B	Tasklet A	Tasklet B	Timer A	Timer B	User Context A	User Context B
IRQ Handler A	None									
IRQ Handler B	SLIS	None								
Softirq A	SLI	SLI	SL							
Softirq B	SLI	SLI	SL	SL						
Tasklet A	SLI	SLI	SL	SL	None					
Tasklet B	SLI	SLI	SL	SL	SL	None				
Timer A	SLI	SLI	SL	SL	SL	SL	None			
Timer B	SLI	SLI	SL	SL	SL	SL	SL	None		
User Context A	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	None	
User Context B	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	MLI	None

Table: Table of Locking Requirements

SLIS	spin_lock_irqsave
SLI	spin_lock_irq
SL	spin_lock
SLBH	spin_lock_bh
MLI	mutex_lock_interruptible

Table: Legend for Locking Requirements Table

The trylock Functions

There are functions that try to acquire a lock only once and immediately return a value telling about success or failure to acquire the lock. They can be used if you need no access to the data protected with the lock when some other thread is holding the lock. You should acquire the lock later if you then need access to the data protected with the lock.

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

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

Common Examples

Let's step through a simple example: a cache of number to name mappings. The cache keeps a count of how often each of the objects is used, and when it gets full, throws out the least used one.

All In User Context

For our first example, we assume that all operations are in user context (ie. from system calls), so we can sleep. This means we can use a mutex to protect the cache and all the objects within it. Here's the code:

```
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/mutex.h>
#include <asm/errno.h>

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

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

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

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

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

/* Must be holding cache_lock */
static void __cache_add(struct object *obj)
{
```

```
        list_add(&obj->list, &cache);
        if (++cache_num > MAX_CACHE_SIZE) {
                struct object *i, *outcast = NULL;
                list_for_each_entry(i, &cache, list) {
                        if (!outcast || i->popularity < outcast->popularity)
                                outcast = i;
                }
                __cache_delete(outcast);
        }
}

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

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

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

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

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

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

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

Note that we always make sure we have the cache_lock when we add, delete, or look up the cache: both the cache infrastructure itself and the contents of the objects are protected by the lock. In this case it's easy, since we copy the data for the user, and never let them access the objects directly.

There is a slight (and common) optimization here: in cache_add() we set up the fields of the object before grabbing the lock. This is safe, as no-one else can access it until we put it in cache.

Accessing From Interrupt Context

Now consider the case where cache_find() can be called from interrupt context: either a hardware interrupt or a softirq. An example would be a timer which deletes object from the cache.

The change is shown below, in standard patch format: the - are lines which are taken away, and the + are lines which are added.

```
--- cache.c.usercontext 2003-12-09 13:58:54.000000000 +1100
+++ cache.c.interrupt   2003-12-09 14:07:49.000000000 +1100
@@ -12,7 +12,7 @@
         int popularity;
 };

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

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

-        mutex_lock(&cache_lock);
+        spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
-        mutex_unlock(&cache_lock);
+        spin_unlock_irqrestore(&cache_lock, flags);
         return 0;
 }

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

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

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

Note that the `spin_lock_irqsave()` will turn off interrupts if they are on, otherwise does nothing (if we are already in an interrupt handler), hence these functions are safe to call from any context.

Unfortunately, `cache_add()` calls `kmalloc()` with the GFP_KERNEL flag, which is only legal in user context. I have assumed that `cache_add()` is still only called in user context, otherwise this should become a parameter to `cache_add()`.

Exposing Objects Outside This File

If our objects contained more information, it might not be sufficient to copy the information in and out: other parts of the code might want to keep pointers to these objects, for example, rather than looking up the id every time. This produces two problems.

The first problem is that we use the cache_lock to protect objects: we'd need to make this non-static so the rest of the code can use it. This makes locking trickier, as it is no longer all in one place.

The second problem is the lifetime problem: if another structure keeps a pointer to an object, it presumably expects that pointer to remain valid. Unfortunately, this is only guaranteed while you hold the lock, otherwise someone might call `cache_delete()` and even worse, add another object, re-using the same address.

As there is only one lock, you can't hold it forever: no-one else would get any work done.

The solution to this problem is to use a reference count: everyone who has a pointer to the object increases it when they first get the object, and drops the reference count when they're finished with it. Whoever drops it to zero knows it is unused, and can actually delete it.

Here is the code:

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

+static void __object_put(struct object *obj)
+{
+        if (--obj->refcnt == 0)
+                kfree(obj);
+}
+
+static void __object_get(struct object *obj)
+{
+        obj->refcnt++;
+}
+
+void object_put(struct object *obj)
+{
+        unsigned long flags;
+
+        spin_lock_irqsave(&cache_lock, flags);
+        __object_put(obj);
+        spin_unlock_irqrestore(&cache_lock, flags);
+}
+
```

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

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

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

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

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

We encapsulate the reference counting in the standard 'get' and 'put' functions. Now we can return the object itself from `cache_find()` which has the advantage that the user can now sleep holding the object (eg. to `copy_to_user()` to name to userspace).

The other point to note is that I said a reference should be held for every pointer to the object: thus the reference count is 1 when first inserted into the cache. In some versions the framework does not hold a reference count, but they are more complicated.

Using Atomic Operations For The Reference Count

In practice, `atomic_t` would usually be used for refcnt. There are a number of atomic operations defined in `include/asm/atomic.h`: these are guaranteed to be seen atomically from all CPUs in the system, so no lock is required. In this case, it is simpler than using spinlocks, although for anything non-trivial using spinlocks is clearer. The `atomic_inc()` and `atomic_dec_and_test()` are used instead of the standard increment and decrement operators, and the lock is no longer used to protect the reference count itself.

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

-static void __object_put(struct object *obj)
-{
-        if (--obj->refcnt == 0)
-                kfree(obj);
-}
-
-static void __object_get(struct object *obj)
-{
-        obj->refcnt++;
-}
-
 void object_put(struct object *obj)
 {
-        unsigned long flags;
-
-        spin_lock_irqsave(&cache_lock, flags);
-        __object_put(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        if (atomic_dec_and_test(&obj->refcnt))
+                kfree(obj);
 }

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

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

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

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

Protecting The Objects Themselves

In these examples, we assumed that the objects (except the reference counts) never changed once they are created. If we wanted to allow the name to change, there are three possibilities:

- You can make cache_lock non-static, and tell people to grab that lock before changing the name in any object.

- You can provide a cache_obj_rename() which grabs this lock and changes the name for the caller, and tell everyone to use that function.

- You can make the cache_lock protect only the cache itself, and use another lock to protect the name.

Theoretically, you can make the locks as fine-grained as one lock for every field, for every object. In practice, the most common variants are:

- One lock which protects the infrastructure (the cache list in this example) and all the objects. This is what we have done so far.

- One lock which protects the infrastructure (including the list pointers inside the objects), and one lock inside the object which protects the rest of that object.

- Multiple locks to protect the infrastructure (eg. one lock per hash chain), possibly with a separate per-object lock.

Here is the "lock-per-object" implementation:

```
--- cache.c.refcnt-atomic   2003-12-11 15:50:54.000000000 +1100
+++ cache.c.perobjectlock   2003-12-11 17:15:03.000000000 +1100
@@ -6,11 +6,17 @@

 struct object
 {
+       /* These two protected by cache_lock. */
        struct list_head list;
+       int popularity;
+
        atomic_t refcnt;
+
+       /* Doesn't change once created. */
        int id;
+
```

```
+        spinlock_t lock; /* Protects the name */
         char name[32];
-        int popularity;
 };

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

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
```

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

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

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

Common Problems

Deadlock: Simple and Advanced

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

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

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

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

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

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

CPU 1	CPU 2
Grab lock A -> OK	Grab lock B -> OK
Grab lock B -> spin	Grab lock A -> spin

Table: Consequences

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

Preventing Deadlock

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

The best locks are encapsulated: they never get exposed in headers, and are never held around calls to non-trivial functions outside the same file. You can read through this code and see that it will never deadlock, because it never tries to grab another lock while it has that one. People using your code don't even need to know you are using a lock.

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

Overzealous Prevention Of Deadlocks

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

If you don't see why, please stay the fuck away from my code.

Racing Timers: A Kernel Pastime

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

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

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

while (list) {
        struct foo *next = list->next;
        del_timer(&list->timer);
        kfree(list);
        list = next;
}

spin_unlock_bh(&list_lock);
```

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

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

```
retry:
        spin_lock_bh(&list_lock);

        while (list) {
                struct foo *next = list->next;
                if (!del_timer(&list->timer)) {
```

```
                        /* Give timer a chance to delete this */
                        spin_unlock_bh(&list_lock);
                        goto retry;
                }
                kfree(list);
                list = next;
        }

        spin_unlock_bh(&list_lock);
```

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

Locking Speed

There are three main things to worry about when considering speed of some code which does locking. First is concurrency: how many things are going to be waiting while someone else is holding a lock. Second is the time taken to actually acquire and release an uncontended lock. Third is using fewer, or smarter locks. I'm assuming that the lock is used fairly often: otherwise, you wouldn't be concerned about efficiency.

Concurrency depends on how long the lock is usually held: you should hold the lock for as long as needed, but no longer. In the cache example, we always create the object without the lock held, and then grab the lock only when we are ready to insert it in the list.

Acquisition times depend on how much damage the lock operations do to the pipeline (pipeline stalls) and how likely it is that this CPU was the last one to grab the lock (ie. is the lock cache-hot for this CPU): on a machine with more CPUs, this likelihood drops fast. Consider a 700MHz Intel Pentium III: an instruction takes about 0.7ns, an atomic increment takes about 58ns, a lock which is cache-hot on this CPU takes 160ns, and a cacheline transfer from another CPU takes an additional 170 to 360ns. (These figures from Paul McKenney's Linux Journal RCU article).

These two aims conflict: holding a lock for a short time might be done by splitting locks into parts (such as in our final per-object-lock example), but this increases the number of lock acquisitions, and the results are often slower than having a single lock. This is another reason to advocate locking simplicity.

The third concern is addressed below: there are some methods to reduce the amount of locking which needs to be done.

Read/Write Lock Variants

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

If your code divides neatly along reader/writer lines (as our cache code does), and the lock is held by readers for significant lengths of time, using these locks can help. They are slightly slower than the normal locks though, so in practice rwlock_t is not usually worthwhile.

Avoiding Locks: Read Copy Update

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

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

```
new->next = list->next;
wmb();
list->next = new;
```

The wmb() is a write memory barrier. It ensures that the first operation (setting the new element's next pointer) is complete and will be seen by all CPUs, before the second operation is (putting the new element into the list). This is important, since modern compilers and modern CPUs can both reorder instructions unless told otherwise: we want a reader to either not see the new element at all, or see the new element with the next pointer correctly pointing at the rest of the list.

Fortunately, there is a function to do this for standard struct list_head lists: list_add_rcu() (include/linux/list.h).

Removing an element from the list is even simpler: we replace the pointer to the old element with a pointer to its successor, and readers will either see it, or skip over it.

```
list->next = old->next;
```

There is list_del_rcu() (include/linux/list.h) which does this (the normal version poisons the old object, which we don't want).

The reader must also be careful: some CPUs can look through the next pointer to start reading the contents of the next element early, but don't realize that the pre-fetched contents is wrong when the next pointer changes underneath them. Once again, there is a list_for_each_entry_rcu() (include/linux/list.h) to help you. Of course, writers can just use list_for_each_entry(), since there cannot be two simultaneous writers.

Our final dilemma is this: when can we actually destroy the removed element? Remember, a reader might be stepping through this element in the list right now: if we free this element and the next pointer changes, the reader will jump off into garbage and crash. We need to wait until we know that all the readers who were traversing the list when we deleted the element are finished. We use call_rcu() to register a callback which will actually destroy the object once all pre-existing readers are finished. Alternatively, synchronize_rcu() may be used to block until all pre-existing are finished.

But how does Read Copy Update know when the readers are finished? The method is this: firstly, the readers always traverse the list inside rcu_read_lock()/rcu_read_unlock() pairs: these simply disable preemption so the reader won't go to sleep while reading the list.

RCU then waits until every other CPU has slept at least once: since readers cannot sleep, we know that any readers which were traversing the list during the deletion are finished, and the callback is triggered. The real Read Copy Update code is a little more optimized than this, but this is the fundamental idea.

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

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

```
+        struct rcu_head rcu;
+
         atomic_t refcnt;

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

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

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

 /* Must be holding cache_lock */
 static void __cache_add(struct object *obj)
 {
-        list_add(&obj->list, &cache);
+        list_add_rcu(&obj->list, &cache);
         if (++cache_num > MAX_CACHE_SIZE) {
                 struct object *i, *outcast = NULL;
                 list_for_each_entry(i, &cache, list) {
@@ -104,12 +114,11 @@
 struct object *cache_find(int id)
 {
         struct object *obj;
-        unsigned long flags;

-        spin_lock_irqsave(&cache_lock, flags);
+        rcu_read_lock();
         obj = __cache_find(id);
         if (obj)
                 object_get(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        rcu_read_unlock();
         return obj;
 }
```

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

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

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

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

The benefit here is that the reference count is not written to: the object is not altered in any way, which is much faster on SMP machines due to caching.

Per-CPU Data

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

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

Of particular use for simple per-cpu counters is the `local_t` type, and the `cpu_local_inc()` and related functions, which are more efficient than simple code on some architectures (`include/asm/local.h`).

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

Data Which Mostly Used By An IRQ Handler

If data is always accessed from within the same IRQ handler, you don't need a lock at all: the kernel already guarantees that the irq handler will not run simultaneously on multiple CPUs.

Manfred Spraul points out that you can still do this, even if the data is very occasionally accessed in user context or softirqs/tasklets. The irq handler doesn't use a lock, and all other accesses are done as so:

```
spin_lock(&lock);
disable_irq(irq);
...
enable_irq(irq);
spin_unlock(&lock);
```

The `disable_irq()` prevents the irq handler from running (and waits for it to finish if it's currently running on other CPUs). The spinlock prevents any other accesses happening at the same time. Naturally, this is slower than just a `spin_lock_irq()` call, so it only makes sense if this type of access happens extremely rarely.

What Functions Are Safe To Call From Interrupts?

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

Some Functions Which Sleep

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

- Accesses to userspace:
    - copy_from_user()
    - copy_to_user()
    - get_user()
    - put_user()
- kmalloc(GFP_KERNEL)
- *mutex_lock_interruptible()* and *mutex_lock()*

    There is a *mutex_trylock()* which does not sleep. Still, it must not be used inside interrupt context since its implementation is not safe for that. *mutex_unlock()* will also never sleep. It cannot be used in interrupt context either since a mutex must be released by the same task that acquired it.

Some Functions Which Don't Sleep

Some functions are safe to call from any context, or holding almost any lock.

- printk()
- kfree()
- add_timer() and del_timer()

Mutex API reference

mutex_init(*mutex*)
    initialize the mutex

Parameters

mutex the mutex to be initialized

Description

Initialize the mutex to unlocked state.

It is not allowed to initialize an already locked mutex.

bool **mutex_is_locked**(struct mutex * *lock*)
    is the mutex locked

Parameters

struct mutex * lock the mutex to be queried

Description

Returns true if the mutex is locked, false if unlocked.

enum mutex_trylock_recursive_enum **mutex_trylock_recursive**(struct mutex * *lock*)
    trylock variant that allows recursive locking

Parameters

struct mutex * lock mutex to be locked

Description

This function should not be used, _ever_. It is purely for hysterical GEM raisins, and once those are gone this will be removed.

Return

- MUTEX_TRYLOCK_FAILED - trylock failed,
- MUTEX_TRYLOCK_SUCCESS - lock acquired,
- MUTEX_TRYLOCK_RECURSIVE - we already owned the lock.

void **mutex_lock**(struct mutex * *lock*)
    acquire the mutex

Parameters

struct mutex * lock the mutex to be acquired

Description

Lock the mutex exclusively for this task. If the mutex is not available right now, it will sleep until it can get it.

The mutex must later on be released by the same task that acquired it. Recursive locking is not allowed. The task may not exit without first unlocking the mutex. Also, kernel memory where the mutex resides must not be freed with the mutex still locked. The mutex must first be initialized (or statically defined) before it can be locked. `memset()`-ing the mutex to 0 is not allowed.

(The CONFIG_DEBUG_MUTEXES .config option turns on debugging checks that will enforce the restrictions and will also do deadlock debugging)

This function is similar to (but not equivalent to) down().

void **mutex_unlock**(struct mutex * *lock*)
    release the mutex

Parameters

struct mutex * lock the mutex to be released

Description

Unlock a mutex that has been locked by this task previously.

This function must not be used in interrupt context. Unlocking of a not locked mutex is not allowed.

This function is similar to (but not equivalent to) up().

void **ww_mutex_unlock**(struct ww_mutex * *lock*)
    release the w/w mutex

Parameters

struct ww_mutex * lock the mutex to be released

Description

Unlock a mutex that has been locked by this task previously with any of the ww_mutex_lock* functions (with or without an acquire context). It is forbidden to release the locks after releasing the acquire context.

This function must not be used in interrupt context. Unlocking of a unlocked mutex is not allowed.

int **mutex_lock_interruptible**(struct mutex * *lock*)
    Acquire the mutex, interruptible by signals.

Parameters

struct mutex * lock The mutex to be acquired.

Description

Lock the mutex like *mutex_lock()*. If a signal is delivered while the process is sleeping, this function will return without acquiring the mutex.

Context

Process context.

Return

0 if the lock was successfully acquired or -EINTR if a signal arrived.

int **mutex_lock_killable**(struct mutex * *lock*)
Acquire the mutex, interruptible by fatal signals.

Parameters

struct mutex * lock The mutex to be acquired.

Description

Lock the mutex like *mutex_lock()*. If a signal which will be fatal to the current process is delivered while the process is sleeping, this function will return without acquiring the mutex.

Context

Process context.

Return

0 if the lock was successfully acquired or -EINTR if a fatal signal arrived.

void **mutex_lock_io**(struct mutex * *lock*)
Acquire the mutex and mark the process as waiting for I/O

Parameters

struct mutex * lock The mutex to be acquired.

Description

Lock the mutex like *mutex_lock()*. While the task is waiting for this mutex, it will be accounted as being in the IO wait state by the scheduler.

Context

Process context.

int **mutex_trylock**(struct mutex * *lock*)
try to acquire the mutex, without waiting

Parameters

struct mutex * lock the mutex to be acquired

Description

Try to acquire the mutex atomically. Returns 1 if the mutex has been acquired successfully, and 0 on contention.

NOTE

this function follows the spin_trylock() convention, so it is negated from the down_trylock() return values! Be careful about this when converting semaphore users to mutexes.

This function must not be used in interrupt context. The mutex must be released by the same task that acquired it.

int **atomic_dec_and_mutex_lock**(atomic_t * *cnt*, struct mutex * *lock*)
return holding mutex if we dec to 0

Parameters

atomic_t * cnt the atomic which we are to dec

struct mutex * lock the mutex to return holding if we dec to 0

Description

return true and hold lock if we dec to 0, return false otherwise

Futex API reference

struct **futex_q**
    The hashed futex queue entry, one per waiting task

Definition

```
struct futex_q {
  struct plist_node list;
  struct task_struct *task;
  spinlock_t *lock_ptr;
  union futex_key key;
  struct futex_pi_state *pi_state;
  struct rt_mutex_waiter *rt_waiter;
  union futex_key *requeue_pi_key;
  u32 bitset;
};
```

Members

list priority-sorted list of tasks waiting on this futex

task the task waiting on the futex

lock_ptr the hash bucket lock

key the key the futex is hashed on

pi_state optional priority inheritance state

rt_waiter rt_waiter storage for use with requeue_pi

requeue_pi_key the requeue_pi target futex key

bitset bitset for the optional bitmasked wakeup

Description

We use this hashed waitqueue, instead of a normal wait_queue_entry_t, so we can wake only the relevant ones (hashed queues may be shared).

A futex_q has a woken state, just like tasks have TASK_RUNNING. It is considered woken when plist_node_empty(q->list) || q->lock_ptr == 0. The order of wakeup is always to make the first condition true, then the second.

PI futexes are typically woken before they are removed from the hash list via the rt_mutex code. See unqueue_me_pi().

struct futex_hash_bucket * **hash_futex**(union futex_key * *key*)
    Return the hash bucket in the global hash

Parameters

union futex_key * key Pointer to the futex key for which the hash is calculated

Description

We hash on the keys returned from get_futex_key (see below) and return the corresponding hash bucket in the global hash.

int **match_futex**(union futex_key * *key1*, union futex_key * *key2*)
    Check whether two futex keys are equal

Parameters

union futex_key * key1 Pointer to key1

union futex_key * key2 Pointer to key2

Description

Return 1 if two futex_keys are equal, 0 otherwise.

int **get_futex_key**(u32 __user * *uaddr*, int *fshared*, union futex_key * *key*, int *rw*)
    Get parameters which are the keys for a futex

Parameters

u32 __user * uaddr virtual address of the futex

int fshared 0 for a PROCESS_PRIVATE futex, 1 for PROCESS_SHARED

union futex_key * key address where result is stored.

int rw mapping needs to be read/write (values: VERIFY_READ, VERIFY_WRITE)

Return

a negative error code or 0

The key words are stored in **key** on success.

For shared mappings, it's (page->index, file_inode(vma->vm_file), offset_within_page). For private mappings, it's (uaddr, current->mm). We can usually work out the index without swapping in the page.

lock_page() might sleep, the caller should not hold a spinlock.

int **fault_in_user_writeable**(u32 __user * *uaddr*)
    Fault in user address and verify RW access

Parameters

u32 __user * uaddr pointer to faulting user space address

Description

Slow path to fixup the fault we just took in the atomic write access to **uaddr**.

We have no generic implementation of a non-destructive write to the user address. We know that we faulted in the atomic pagefault disabled section so we can as well avoid the #PF overhead by calling get_user_pages() right away.

struct *futex_q* * **futex_top_waiter**(struct futex_hash_bucket * *hb*, union futex_key * *key*)
    Return the highest priority waiter on a futex

Parameters

struct futex_hash_bucket * hb the hash bucket the futex_q's reside in

union futex_key * key the futex key (to distinguish it from other futex futex_q's)

Description

Must be called with the hb lock held.

int **futex_lock_pi_atomic**(u32 __user * *uaddr*, struct futex_hash_bucket * *hb*, union futex_key * *key*, struct futex_pi_state ** *ps*, struct task_struct * *task*, int *set_waiters*)
    Atomic work required to acquire a pi aware futex

Parameters

u32 __user * uaddr the pi futex user address

struct futex_hash_bucket * hb the pi futex hash bucket

union futex_key * key the futex key associated with uaddr and hb

struct futex_pi_state ** ps the pi_state pointer where we store the result of the lookup

struct task_struct * task the task to perform the atomic lock work for. This will be "current" except in the case of requeue pi.

int set_waiters force setting the FUTEX_WAITERS bit (1) or not (0)

Return

- 0 - ready to wait;
- 1 - acquired the lock;
- <0 - error

The hb->lock and futex_key refs shall be held by the caller.

void **__unqueue_futex**(struct *futex_q * q*)
    Remove the futex_q from its futex_hash_bucket

Parameters

struct futex_q * q The futex_q to unqueue

Description

The q->lock_ptr must not be NULL and must be held by the caller.

void **requeue_futex**(struct *futex_q * q*, struct futex_hash_bucket * *hb1*, struct futex_hash_bucket
                * *hb2*, union futex_key * *key2*)
    Requeue a futex_q from one hb to another

Parameters

struct futex_q * q the futex_q to requeue

struct futex_hash_bucket * hb1 the source hash_bucket

struct futex_hash_bucket * hb2 the target hash_bucket

union futex_key * key2 the new key for the requeued futex_q

void **requeue_pi_wake_futex**(struct *futex_q * q*, union futex_key * *key*, struct futex_hash_bucket
                        * *hb*)
    Wake a task that acquired the lock during requeue

Parameters

struct futex_q * q the futex_q

union futex_key * key the key of the requeue target futex

struct futex_hash_bucket * hb the hash_bucket of the requeue target futex

Description

During futex_requeue, with requeue_pi=1, it is possible to acquire the target futex if it is uncontended or via a lock steal. Set the futex_q key to the requeue target futex so the waiter can detect the wakeup on the right futex, but remove it from the hb and NULL the rt_waiter so it can detect atomic lock acquisition. Set the q->lock_ptr to the requeue target hb->lock to protect access to the pi_state to fixup the owner later. Must be called with both q->lock_ptr and hb->lock held.

int **futex_proxy_trylock_atomic**(u32 __user * *pifutex*, struct futex_hash_bucket * *hb1*, struct futex_hash_bucket * *hb2*, union futex_key * *key1*, union futex_key
                        * *key2*, struct futex_pi_state ** *ps*, int *set_waiters*)
    Attempt an atomic lock for the top waiter

Parameters

u32 __user * pifutex the user address of the to futex

struct futex_hash_bucket * hb1 the from futex hash bucket, must be locked by the caller

struct futex_hash_bucket * hb2 the to futex hash bucket, must be locked by the caller

union futex_key * key1 the from futex key

union futex_key * key2 the to futex key

struct futex_pi_state ** ps address to store the pi_state pointer

int set_waiters force setting the FUTEX_WAITERS bit (1) or not (0)

Description

Try and get the lock on behalf of the top waiter if we can do it atomically. Wake the top waiter if we succeed. If the caller specified set_waiters, then direct *futex_lock_pi_atomic()* to force setting the FUTEX_WAITERS bit. hb1 and hb2 must be held by the caller.

Return

- 0 - failed to acquire the lock atomically;
- >0 - acquired the lock, return value is vpid of the top_waiter
- <0 - error

int **futex_requeue**(u32 __user * *uaddr1*, unsigned int *flags*, u32 __user * *uaddr2*, int *nr_wake*, int *nr_requeue*, u32 * *cmpval*, int *requeue_pi*)
    Requeue waiters from uaddr1 to uaddr2

Parameters

u32 __user * uaddr1 source futex user address

unsigned int flags futex flags (FLAGS_SHARED, etc.)

u32 __user * uaddr2 target futex user address

int nr_wake number of waiters to wake (must be 1 for requeue_pi)

int nr_requeue number of waiters to requeue (0-INT_MAX)

u32 * cmpval uaddr1 expected value (or NULL)

int requeue_pi if we are attempting to requeue from a non-pi futex to a pi futex (pi to pi requeue is not supported)

Description

Requeue waiters on uaddr1 to uaddr2. In the requeue_pi case, try to acquire uaddr2 atomically on behalf of the top waiter.

Return

- >=0 - on success, the number of tasks requeued or woken;
- <0 - on error

void **queue_me**(struct *futex_q* * *q*, struct futex_hash_bucket * *hb*)
    Enqueue the futex_q on the futex_hash_bucket

Parameters

struct futex_q * q The futex_q to enqueue

struct futex_hash_bucket * hb The destination hash bucket

Description

The hb->lock must be held by the caller, and is released here. A call to *queue_me()* is typically paired with exactly one call to *unqueue_me()*. The exceptions involve the PI related operations, which may use

unqueue_me_pi() or nothing if the unqueue is done as part of the wake process and the unqueue state is implicit in the state of woken task (see *futex_wait_requeue_pi()* for an example).

int **unqueue_me**(struct *futex_q* * *q*)
    Remove the futex_q from its futex_hash_bucket

Parameters

struct futex_q * q The futex_q to unqueue

Description

The q->lock_ptr must not be held by the caller. A call to *unqueue_me()* must be paired with exactly one earlier call to *queue_me()*.

Return

   • 1 - if the futex_q was still queued (and we removed unqueued it);

   • 0 - if the futex_q was already removed by the waking thread

int **fixup_owner**(u32 __user * *uaddr*, struct *futex_q* * *q*, int *locked*)
    Post lock pi_state and corner case management

Parameters

u32 __user * uaddr user address of the futex

struct futex_q * q futex_q (contains pi_state and access to the rt_mutex)

int locked if the attempt to take the rt_mutex succeeded (1) or not (0)

Description

After attempting to lock an rt_mutex, this function is called to cleanup the pi_state owner as well as handle race conditions that may allow us to acquire the lock. Must be called with the hb lock held.

Return

   • 1 - success, lock taken;

   • 0 - success, lock not taken;

   • <0 - on error (-EFAULT)

void **futex_wait_queue_me**(struct futex_hash_bucket * *hb*, struct *futex_q* * *q*, struct hrtimer_sleeper * *timeout*)
    queue_me() and wait for wakeup, timeout, or signal

Parameters

struct futex_hash_bucket * hb the futex hash bucket, must be locked by the caller

struct futex_q * q the futex_q to queue up on

struct hrtimer_sleeper * timeout the prepared hrtimer_sleeper, or null for no timeout

int **futex_wait_setup**(u32 __user * *uaddr*, u32 *val*, unsigned int *flags*, struct *futex_q* * *q*, struct futex_hash_bucket ** *hb*)
    Prepare to wait on a futex

Parameters

u32 __user * uaddr the futex userspace address

u32 val the expected value

unsigned int flags futex flags (FLAGS_SHARED, etc.)

struct futex_q * q the associated futex_q

struct futex_hash_bucket ** hb storage for hash_bucket pointer to be returned to caller

Description

Setup the futex_q and locate the hash_bucket. Get the futex value and compare it with the expected value. Handle atomic faults internally. Return with the hb lock held and a q.key reference on success, and unlocked with no q.key reference on failure.

Return

- 0 - uaddr contains val and hb has been locked;
- <1 - -EFAULT or -EWOULDBLOCK (uaddr does not contain val) and hb is unlocked

int **handle_early_requeue_pi_wakeup**(struct futex_hash_bucket * *hb*, struct *futex_q* * *q*, union futex_key * *key2*, struct hrtimer_sleeper * *timeout*)
      Detect early wakeup on the initial futex

Parameters

struct futex_hash_bucket * hb the hash_bucket futex_q was original enqueued on

struct futex_q * q the futex_q woken while waiting to be requeued

union futex_key * key2 the futex_key of the requeue target futex

struct hrtimer_sleeper * timeout the timeout associated with the wait (NULL if none)

Description

Detect if the task was woken on the initial futex as opposed to the requeue target futex. If so, determine if it was a timeout or a signal that caused the wakeup and return the appropriate error code to the caller. Must be called with the hb lock held.

Return

- 0 = no early wakeup detected;
- <0 = -ETIMEDOUT or -ERESTARTNOINTR

int **futex_wait_requeue_pi**(u32 __user * *uaddr*, unsigned int *flags*, u32 *val*, ktime_t * *abs_time*, u32 *bitset*, u32 __user * *uaddr2*)
      Wait on uaddr and take uaddr2

Parameters

u32 __user * uaddr the futex we initially wait on (non-pi)

unsigned int flags futex flags (FLAGS_SHARED, FLAGS_CLOCKRT, etc.), they must be the same type, no requeueing from private to shared, etc.

u32 val the expected value of uaddr

ktime_t * abs_time absolute timeout

u32 bitset 32 bit wakeup bitset set by userspace, defaults to all

u32 __user * uaddr2 the pi futex we will take prior to returning to user-space

Description

The caller will wait on uaddr and will be requeued by *futex_requeue()* to uaddr2 which must be PI aware and unique from uaddr. Normal wakeup will wake on uaddr2 and complete the acquisition of the rt_mutex prior to returning to userspace. This ensures the rt_mutex maintains an owner when it has waiters; without one, the pi logic would not know which task to boost/deboost, if there was a need to.

We call schedule in *futex_wait_queue_me()* when we enqueue and return there via the following– 1) wakeup on uaddr2 after an atomic lock acquisition by *futex_requeue()* 2) wakeup on uaddr2 after a requeue 3) signal 4) timeout

If 3, cleanup and return -ERESTARTNOINTR.

If 2, we may then block on trying to take the rt_mutex and return via: 5) successful lock 6) signal 7) timeout 8) other lock acquisition failure

If 6, return -EWOULDBLOCK (restarting the syscall would do the same).

If 4 or 7, we cleanup and return with -ETIMEDOUT.

Return

- 0 - On success;
- <0 - On error

long **sys_set_robust_list**(struct robust_list_head __user * *head*, size_t *len*)
    Set the robust-futex list head of a task

Parameters

struct robust_list_head __user * head pointer to the list-head

size_t len length of the list-head, as userspace expects

long **sys_get_robust_list**(int *pid*, struct robust_list_head __user *__user * *head_ptr*, size_t __user
    * *len_ptr*)
    Get the robust-futex list head of a task

Parameters

int pid pid of the process [zero for current task]

struct robust_list_head __user *__user * head_ptr pointer to a list-head pointer, the kernel fills
    it in

size_t __user * len_ptr pointer to a length field, the kernel fills in the header size

Further reading

- `Documentation/locking/spinlocks.txt`: Linus Torvalds' spinlocking tutorial in the kernel sources.
- Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers:

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

Thanks

Thanks to Telsa Gwynne for DocBooking, neatening and adding style.

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

Thanks to the cabal for having no influence on this document.

Glossary

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

bh Bottom Half: for historical reasons, functions with '_bh' in them often now refer to any software interrupt, e.g. `spin_lock_bh()` blocks any software interrupt on the current CPU. Bottom halves are deprecated, and will eventually be replaced by tasklets. Only one bottom half will be running at any time.

Hardware Interrupt / Hardware IRQ Hardware interrupt request. `in_irq()` returns true in a hardware interrupt handler.

Interrupt Context Not user context: processing a hardware irq or software irq. Indicated by the `in_interrupt()` macro returning true.

SMP Symmetric Multi-Processor: kernels compiled for multiple-CPU machines. (`CONFIG_SMP=y`).

Software Interrupt / softirq Software interrupt handler. `in_irq()` returns false; `in_softirq()` returns true. Tasklets and softirqs both fall into the category of 'software interrupts'.

Strictly speaking a softirq is one of up to 32 enumerated software interrupts which can run on multiple CPUs at once. Sometimes used to refer to tasklets as well (ie. all software interrupts).

tasklet A dynamically-registrable software interrupt, which is guaranteed to only run on one CPU at a time.

timer A dynamically-registrable software interrupt, which is run at (or close to) a given time. When running, it is just like a tasklet (in fact, they are called from the `TIMER_SOFTIRQ`).

UP Uni-Processor: Non-SMP. (`CONFIG_SMP=n`).

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

Userspace A process executing its own code outside the kernel.