Guía Informal al Bloqueo

Paul Rusty Russell

rusty@rustcorp.com.au

Guía Informal al Bloqueo

por Paul Rusty Russell

Tabla de contenidos

Lista de tablas

Capítulo 1. Introducción

Bienvenido, a la Guía Informal de Bloqueo de Núcleo de Rusty. Este documento describe los sistemas de bloqueo en el núcleo Linux como aproximación al 2.4.

Parece que es aquí donde tiene que estar *SMP* ; por lo tanto todo el mundo que esté en estos días hackeando el núcleo necesita conocer los fundamentos de la concurrencia y el bloqueos para SMP.

1.1. El Problema con la Concurrencia

(Sáltate esto si sabes lo que es una Condición de Carrera (Race Condition).

En un programa normal, puedes incrementar un contador de la forma:

```
contador_muy_importante++;
```

Esto es lo que esperarías que pasase:

Tabla 1-1. Resultados Esperados

Instancia 1	Instancia 2
lee contador_muy_importante (5)	
añade 1 (6)	
escribe contador_muy_importante (6)	
	lee contador_muy_importante (6)
	añade 1 (7)
	escribe contador_muy_importante (7)

Esto es lo que quizás pase:

Tabla 1-2. Resultados Posibles

Instancia 1	Instancia 2
lee contador_muy_importante (5)	
	lee contador_muy_importante (5)
añade 1 (6)	
	añade 1 (6)
escribe contador_muy_importante (6)	
	escribe contador_muy_importante (6)

Este solapamiento, donde lo que sucede depende del tiempo relativo de múltiples tareas, es llamado condición de carrera. La parte de código que contenie al punto de concurrencia se llamado región crítica. Y especialmente desde que Linux se empezó a ejecutar en máquinas SMP, se ha convertido en uno de los puntos más grandes del diseño e implementación del núcleo.

La solución es reconocer cuando ocurren estos accesos simultáneos, y usar bloqueos para asegurar que sólo una instancia puede entrar en la región crítica en cada instante. Hay muchas primitivas amigables en el núcleo Linux que te ayudan a hacer esto. Y entonces hay primitivas no amigables, pero yo intento que no existan.

Capítulo 2. Dos Tipos Principales de Bloqueos del Núcleo: Spinlocks y Semáforos

Hay dos tipos principales de bloqueos del núcleo. El tipo fundamental es el spinlock (`include/asm/spinlock.h`), que es un bloqueo muy simple receptáculo-simple; si no puedes coger el spinlock, entonces te mantienes intentándolo (spinning) hasta que puedas. Los spinlocks son muy pequeños y rápidos, y pueden ser usados en cualquier sitio.

El segundo tipo es el semáforo (`include/asm/semaphore.h`): puede tener más de un receptáculo en algún momento (el número se decide en tiempo de inicialización), aunque es usado más comúnmente como un bloqueo de receptáculo-simple (un mutex). Si no puedes obtener el semáforo, tus tareas se pondrán en una cola, y serán despertadas cuando el semáforo sea liberado. Esto significa que la CPU hará algo mientras que estás esperando, pero hay muchos casos en los que simplemente no puedes dormir (ver Sección 4.8), y por lo tanto tienes que usar un spinlock en vez del semáforo.

Ningún tipo de bloqueo es recursivo: ver Sección 4.2.

2.1. Bloqueos y Núcleos Monoprocesador

Para núcleos compilados sin CONFIG_SMP, los spinlocks no existen. Esta es una excelente decisión de diseño; cuando nadie se puede ejecutar al mismo tiempo, no hay motivo para tener un bloqueo.

Deberías siempre de probar tu código de bloqueo con CONFIG_SMP habilitado, incluso si no tienes un equipo de prueba SMP, porque aún así pillará algunos tipos (simples) de deadlock.

Los semáforos todavía existen, porque son requeridos para la sincronización entre *contextos de usuario*, tal como veremos a continuación.

2.2. Variantes de Bloqueo Lectura/Escritura

Los spinlocks y los semáforos tienen variantes de lectura/escritura: rwlock_t y struct rw_semaphore. Estos dividen a los usuarios en dos clases: los lectores y los escritores. Si sólo estás leyendo datos, puedes coger un bloqueo de lectura, pero para escribir los datos necesitas un bloqueo de escritura. Mucha gente puede tener un bloqueo de lectura, pero uno de escritura debe de ser único.

Esto significa que es mucho más fácil bloquear si tu código se divide ordenadamente entre líneas lectoras y escritoras. Toda las discusiones posteriores también se aplican a las variantes de lectura/escritura.

2.3. Bloqueando Sólo en el Contexto de Usuario

Si tienes una estructura de datos que siempre es accedida desde el contexto de usuario, entonces puedes usar un semáforo simple (`linux/asm/semaphore.h`) para protegerla. Este es el caso más trivial; inicializas el semáforo al número de recursos disponibles (usualmente 1), y llamas a `down_interruptible()` para coger el semáforo, y `up()` para liberarlo. Hay también una función `down()`, que debería de ser evitada, porque no regresará si se recibe una señal.

Ejemplo: `linux/net/core/netfilter.c` permite el registro de unas nuevas llamadas `setsockopt()` y `getsockopt()`. El registro y desregistro sólo son realizadas en la carga y descarga de un módulo (y tiempo de arranque, donde no hay concurrencia), y la lista de registros sólo es consultada por una llamada al sistema desconocida `setsockopt()` o `getsockopt()`. La `nf_sockopt_mutex` es perfecta para proteger esto, especialmente desde que las llamadas setsockopt y getsockopt quizás se vayan a dormir. sleep.

2.4. Bloqueando entre Contexto de Usuario y BHs (Bottom Halves)

Si un *bottom half* comparte datos con el contexto de usuario, tienes dos problemas. El primero, el actual contexto de usuario puede ser interrumpido por un bottom half, y el segundo, la región crítica puede ser ejecutada desde otra CPU. Aquí es donde es usado `spin_lock_bh()` (`include/linux/spinlock.h`). El deshabilita los bottom halves es esta CPU, entonces coge el bloqueo. `spin_unlock_bh()` realiza lo inverso.

Esto además funciona perfectamente para *UP* ; el spinlock desaparece, y esta macro simplemente se transforma en `local_bh_disable()` (`include/asm/softirq.h`), la cual te protege de que el bottom half se ejecute.

2.5. Bloqueando Entre Contexto de Usuario y Tasklets/Soft IRQs

Esto es exactamente lo mismo que lo anterior, porque `local_bh_disable()` actualmente también deshabilita todas las softirqs y *tasklets* en esta CPU. Debería de ser llamada ‘local_softirq_disable()’, pero el nombre ha sido preservado por motivos históricos. De forma similar, en un mundo perfecto `spin_lock_bh()` debería de ser llamada spin_lock_softirq().

2.6. Bloqueando Entre Bottom Halves

Algunas veces un bottom half quizás quiera compartir datos con otro bottom half (recuerda especialmente que los cronómetros se ejecutan en un bottom half).

2.6.1. El Mismo BH

Como un bottom half nunca se ejecutará en dos CPUs a la vez, no necesitas preocuparte sobre que tu bottom half se encuentre ejecutando dos veces al mismo tiempo, incluso en SMP.

2.6.2. Diferentes BHs

Como sólo un bottom half se ejecuta en un mismo instante, no necesitas preocuparte sobre las condiciones de carrera con otros bottom halves. Cuidate de las cosas que quizás cambien debajo de ti, por ejemplo, si alguien cambia tu bottom half a una tasklet. Si quieres hacer tu código preparado para el futuro, finge que ya te estás ejecutando desde una tasklet (ver después), y haz el bloqueo extra. Por supuesto, si esto es cinco años antes de que ocurra parecerás una maldición tonta.

2.7. Bloqueando Entre Tasklets

Algunas veces una tasklet quizás quiera compartir datos con otra tasklet, o con un bottom half.

2.7.1. La Misma Tasklet

Como una tasklet nunca se ejecutará en dos CPUs al mismo tiempo, no tienes que preocuparte sobre que tu tasklet sea reentrante (ejecutándose dos veces al mismo tiempo), incluso en SMP.

2.7.2. Diferentes Tasklets

Si otra tasklet (o bottom half, tales como cronómetros) quiere compartir datos con tu tasklet, necesitarás usar las llamadas `spin_lock()` y `spin_unlock()`. `spin_lock_bh()` es innecesaria aquí, tal y como ya has visto en una tasklet, y ninguna será ejecutada en la misma CPU,

2.8. Bloqueando entre Softirqs

Frecuentemente una *softirq* quizás quiera compartir datos con ella misma, con una tasklet, o con un bottom half.

2.8.1. La Misma Softirq

La misma softirq puede ejecutarse en otras CPUs: puedes usar un array para cada CPU (ver Sección 4.3) para un mejor rendimiento. Si vas a llegar tan lejos como el uso de una softirq, probablemente te

preocupes suficientemente sobre el rendimiento escalable para justificar la complejidad extra.

Necesitarás usar `spin_lock()` y `spin_unlock()` para compartir datos.

2.8.2. Diferentes Softirqs

Necesitarás usar `spin_lock()` y `spin_unlock()` para datos compartidos, cuando sea un cronómetro (que puede ejecutarse en una CPU diferente), bottom halt, tasklet o la misma u otra softirq.

Capítulo 3. Contexto de IRQ de Hardware

Las interrupciones hardware usualmente se comunican con un bottom half, tasklet o softirq. Frecuentemente esto complica el poner el trabajo en una cola, que el BH/softirq debería de sacar.

3.1. Bloqueando entre IRQs Hardware y Softirqs/Tasklets/BHs

Si un manejador irq hardware comparte datos con una softirq, tienes dos problemas. Primeramente, la softirq procesando puede ser interrumpida por una interrupción hardware, y segundo, la región crítica podría ser entrada por una interrupción hardware en otra CPU. Aquí es donde se usa `spin_lock_irq()`. Está definida para deshabilitar las interrupciones en esa cpu, entonces coge el bloqueo. `spin_unlock_irq()` hace lo inverso.

Esto también trabaja perfectamente para UP: el spinlock se desvanece, y esta macro simplemente se convierte en `local_irq_disable()` (`include/asm/smp.h`), qye te protege de que las softirq/tasklet/BH se ejecuten.

`spin_lock_irqsave()` (`include/linux/spinlock.h`) es una variante que salva cuando las interrupciones estaban habilidatas o deshabilitadas en una palabra de flags, que es pasada a `spin_lock_irqrestore()`. Esto significa que el mismo código puede ser usado dentro de un manejador irq hardware (donde las interrupciones ya estan deshabilitadas) y en softirqs (donde se requiere el deshabilitar las irqs).

Capítulo 4. Técnicas Comunes

Esta sección lista algunos dilemas comunes y las soluciones estándar usadas en el código del núcleo Linux. Si usas estas, la gente encontrará tu código más fácil de entender.

Si pudiera darte una parte de un aviso sería: nunca duermas con alguien más loco/a que tú. Pero si tuviera que darte un aviso en el bloqueo: *mantente sólo*.

Bloquea a los datos, no al código.

Se reacio a introducir nuevos bloqueos.

Suficientemente ajeno, esto es justo lo contrario de mi aviso cuando *tienes* que dormir con alguien más loco/a que tú.

4.1. En un Contexto de Interrupciones No Escritores

Hay un caso bastante común donde un manejador de interrupciones necesita acceder a la región crítica, pero no necesita acceso de escritura. En este caso, no necesitas usar `read_lock_irq()`, únicamente `read_lock()` en todos los sitios (desde que ocurre una interrupción, el manejador irq sólo intentará coger el bloqueo, que no hará deadlock). Todavía necesitas usar `write_lock_irq()`.

Una lógica similar se aplica al bloqueo entre softirqs/tasklets/BHs que nunca necesitan un bloqueo de escritura, y al contexto de usuario: `read_lock()` y `write_lock_bh()`.

4.2. Deadlock: Simple y Avanzado

Hay un fallo de codificación donde un pedazo de código intenta obtener un spinlock dos veces: él esperará siempre, esperando a que el bloqueo sea liberado (spinlocks, rwlocks y semáforos no son recursivos en Linux). Esto es trivial de diagnosticar: no es un tipo de problema de estar-cinco-noches-despierto-hablando-con-los-suaves-conejitos-del-código.

Para un caso ligeramente más complejo, imagínate que tienes una región compartida por un bottom half y un contexto de usuario. Si usas una llamada `spin_lock()` para protegerla, es posible que el contexto de usuario sea interrumpido por el bottom half mientras mantiene el bloqueo, y el bottom half entonces esperará para siempre para obtener el mismo bloqueo.

Ambas son llamadas deadlock (bloqueo muerto), y como se mostró antes, puede ocurrir con una CPU simple (aunque no en compilaciones para UP, ya que los spinlocks se desvanecen en la compilación del

núcleo con CONFIG_SMP=n. Aún tendrás corrupción de datos en el segundo ejemplo).

Este bloqueo completo es fácil de diagnosticar: en equipos SMP el cronómetro guardián o compilado con DEBUG_SPINLOCKS establecido (`include/linux/spinlock.h`) nos mostrará esto inmediatamente cuando suceda.

Un problema más complejo es el también llamado ‘abrazo mortal’, involucrando a dos o más bloqueos. Digamos que tienes una tabla hash: cada entrada en la tabla es un spinlock, y una cadena de objetos ordenados. Dentro de un manejador softirq, algunas veces quieres alterar un objeto de un lugar de la tabla hash a otro: coges el spinlock de la vieja cadena hash y el spinlock de la nueva cadena hash, y borras el objeto de la vieja y lo insertas en la nueva.

Aquí hay dos problemas. El primero es que si tu código siempre intenta mover el objeto a la misma cadena, él se hará un deadlock cuando se intente bloquear dos veces. El segundo es que si la misma softirq u otra CPU está intentando mover otro objeto en la dirección inversa podría pasar lo siguiente:

Tabla 4-1. Consecuencias

CPU 1	CPU 2
Pilla bloqueo A -> OK	Pilla bloqueo B -> OK
Pilla bloqueo B -> spin	Pilla bloqueo A -> spin

Las dos CPUs esperarán para siempre, esperando a que el otro libere su bloqueo. Él parecerá, olerá, y se sentirá como si cayera el sistema.

4.2.1. Preveniendo los Deadlocks

Los libros de texto te dirán que si siempre bloqueas en el mismo orden, nunca obtendrás esta clase de deadlock. La práctica te dirá que este tipo de aproximación no escala bien: cuando creo un nuevo bloqueo, no entiendo suficientemente el núcleo para imaginarme dónde está él en la jerarquía de los 5000 bloqueos.

Los mejores bloqueos están encapsulados; nunca estarán expuestos en las cabeceras, y nunca se mantendrán a través de llamadas a funciones no triviales fuera del mismo archivo. Puedes leer a través de este código y ver que nunca hará deadlock, porque nunca intenta tener otro bloqueo mientras tiene el uso. La gente usando tu código no necesita saber nunca que estás usando un bloqueo.

Un problema clásico aquí es cuando suministras retrollamadas o trampas: si las llamas con el bloqueo mantenido, arriesgas un deadlock simple, o un abrazo mortal (¿quién sabe lo que hará la llamada?). Recuerda, los otros programadores andan detrás de ti, por lo tanto no hagas esto.

4.2.2. Sobreentusiasmo en la Prevención de Deadlocks

Los deadlocks son problemáticos, pero no son tan malos como la corrupción de datos. El código que obtiene un bloqueo de lectura, busca una lista, falla al encontrar lo que quiere, tira el bloqueo de lectura, obtiene un bloqueo de escritura e inserta el objeto tiene una condición de carrera.

Si no ves porqué, por favor permanece jodidamente lejos de mi código.

4.3. Datos por cada CPU

Una gran técnica usada ampliamente para eliminar el bloqueo es duplicar la información para cada CPU. Por ejemplo, si quieres mantener una cuenta de una condición común, puedes usar un spinlock y un contador simple. Bonito y simple.

Si esto era muy lento [probablemente no], puedes en vez de esto usar un contador para cada CPU [no lo hagas], entonces ninguno de ellos necesitarán un bloqueo exclusivo [estás gastando tu tiempo aquí]. Para asegurarte de que las CPUs no tienen que sincronizar las cachés todo el tiempo, alinea los contadores al límite de las cachés añadiendo '__cacheline_aligned' a la declaración (`include/linux/cache.h`). [¿No puedes pensar en alguna cosa mejor que hacer?]

De cualquier forma necesitarán un bloqueo de lectura para acceder a sus propios contadores. De esta forma puedes usar un bloqueo de escritura para garantizar acceso exclusivo a todos ellos a la vez, para llevar cuenta de ellos.

4.4. Bloqueos Gran Lector

Un ejemplo clásico de información para cada CPU son los bloqueos 'gran lector' de Ingo (`linux/include/brlock.h`). Estos usan técnicas de datos de cada CPU descritas más adelante para crear un bloqueo que es muy rápido en obtener un bloqueo de lectura, pero agonizantemente lento para un bloqueo de escritura.

Afortunadamente, hay un número limitado disponible de estos bloqueos: tienes que ir a través de un proceso de entrevista estricta para obtener uno.

4.5. Eliminando los bloqueos: Ordenamiento de Lecturas y Escrituras

Algunas veces es posible eliminar el bloqueo. Considera el siguiente caso del código del cortafuegos 2.2, que inserta un elemento en una lista simplemente enlazada en el contexto de usuario:

```
new->next = i->next;
i->next = new;
```

Aquí el autor (Alan Cox, que sabía lo que estaba haciendo) asume que el asignamiento de punteros es atómico. Esto es importante, porque los paquetes de red atravesarían esta lista en bottom halves sin un bloqueo. Dependiendo del tiempo exacto, ellos verían el nuevo elemento en las lista con un puntero *`next`* válido, o no verían la lista todavía. Aún se requiere un bloqueo contra otras CPUs insertando o borrando de la lista, por supuesto.

Por supuesto, las escrituras *deben* estar en este orden, en otro caso el nuevo elemento aparece en la lista con un puntero *`next`* inválido, y alguna otra CPU iterando en el tiempo equivocado saltará a través de él a la basura. Porque las modernas CPUs reordenan, el código de Alan actualmente se lee como sigue:

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

La función `wmb()` es una barrera de escritura de memoria (`include/asm/system.h`): ni el compilador ni la CPU permitirán alguna escritura a memoria después de que `wmb()` sea visible a otro hardware antes de que alguna otra escritura se encuentre antes de `wmb()`.

Como i386 no realiza reordenamiento de escritura, este bug nunca fue mostrada en esta plataforma. Es otras plataformas SMP, de cualquier forma, si que fue mostrado.

También hay `rmb()` para ordenamiento de lectura: para asegurar que cualquier lectura previa de una variable ocurre antes de la siguiente lectura. La macro simple `mb()` combina `rmb()` y `wmb()`.

Algunas operaciones atómicas están definidas para actuar como una barrera de memoria (esto es, como la macro `mb()`, pero si dudas, se explícito. También, las operaciones de spinlock actuan como barreras parciales: las operaciones después de obtener un spinlock nunca serán movidas para preceder a la llamada `spin_lock()`, y las operaciones antes de liberar un spinlock nunca serán movidas después de la llamada `spin_unlock()`.

4.6. Eliminando los Bloqueos: Operaciones Atómicas

Hay un número de operaciones atómicas definidas en `include/asm/atomic.h`: estas están garantizadas que serán atómicas para todas las CPUs en el sistema, entonces eliminando las carreras. Si tus datos compartidos consisten, digamos, en un simple contador, estas operaciones quizás sean más simples que usar spinlocks (aunque para algo no trivial el uso de spinlocks es más claro).

Destacar que las operaciones atómicas están definidas para actuar como barreras de escritura y lectura en todas las plataformas.

4.7. Protegiendo Una Colección de Objetos: Cuentas de Referencia

Bloqueando una colección de objetos es bastante fácil: coges un spinlock simple, y te aseguras de obtenerlo antes de buscar, añadir o borrar un objeto.

El propósito de este bloqueo no es proteger los objetos individuales: quizás tengas un bloqueo separado dentro de cada uno de ellos. Es para proteger la *estructura de datos conteniendo el objeto* de las condiciones de carrera. Frecuentemente el mismo bloqueo es usado también para proteger los contenidos de todos los objetos, por simplicidad, pero ellos son inherentemente ortogonales (y muchas otras grandes palabras diseñadas para confundir).

Cambiando esto a un bloqueo de lectura-escritura frecuentemente ayudará notablemente si las lecturas son más frecuentes que las escrituras. Si no, hay otra aproximación que puedes usar para reducir el tiempo que es mantenido el bloqueo: las cuentas de referencia.

En esta aproximación, un objeto tiene un dueño, quien establece la cuenta de referencia a uno. Cuando obtienes un puntero al objeto, incrementas la cuenta de referencia (una operación ’obtener’). Cuando abandonas un puntero, decrementas la cuenta de referencia (una operación ’poner’). Cuando el dueño quiere destruirlo, lo marca como muerto y hace una operación poner.

Cualquiera que ponga la cuenta de referencia a cero (usualmente implementado con `atomic_dec_and_test()`) limpia y libera el objeto.

Esto significa que se garantiza que el objeto no se desvanecerá debajo de ti, incluso aunque no tengas más un bloqueo para la colección.

Aquí hay algún código esqueleto:

```
        void create_foo(struct foo *x)
        {
                atomic_set(&x->use, 1);
                spin_lock_bh(&list_lock);
                ... inserta en la lista ...
                spin_unlock_bh(&list_lock);
        }

        struct foo *get_foo(int desc)
        {
                struct foo *ret;

                spin_lock_bh(&list_lock);
                ... encuentra en la lista ...
                if (ret) atomic_inc(&ret->use);
                spin_unlock_bh(&list_lock);

                return ret;
```

```
}

void put_foo(struct foo *x)
{
        if (atomic_dec_and_test(&x->use))
                kfree(foo);
}

void destroy_foo(struct foo *x)
{
        spin_lock_bh(&list_lock);
        ... borra de la lista ...
        spin_unlock_bh(&list_lock);

        put_foo(x);
}
```

4.7.1. Macros Para Ayudarte

Hay un conjunto de macros de depuración recogidas dentro de `include/linux/netfilter_ipv4/lockhelp.h` y `listhelp.h`: estas son muy útiles para asegurarnos de que los bloqueos son mantenidos en los sitios correctos para proteger la infraestructura.

4.8. Cosas Que Duermen

Nunca puedes llamar a las siguientes rutinas mientras estás manteniendo un spinlock, porque ellas quizás se vayan a dormir. Esto también significa que necesitas estar en el contexto de usuario.

- Accesos a *userspace*:
  - `copy_from_user()`
  - `copy_to_user()`
  - `get_user()`
  - `put_user()`
- `kmalloc(GFP_KERNEL)`
- `down_interruptible()` y `down()`

  Hay una función `down_trylock()` que puede ser usada dentro del contexto de interrupción, ya que no dormirá. `up()` tampoco dormirá.

`printk()` puede ser llamada en *cualquier* contexto, suficientemente interesante.

4.9. La Follonera Sparc

Alan Cox dice “la deshabilitación/habilitación de una sparc es en la ventana registrada”. Andi Kleen dice “cuando restore_flags (restauras las banderas) en una función diferente ensucias todas las ventanas de registros”.

Por lo tanto nunca pases el conjunto de palabras de flags por `spin_lock_irqsave()` y hermanos a otra función (a menos que sea declarada inline). Usualmente nadie hace esto, pero ahora ya estás advertido. Dave Miller nunca puede hacer nada de una forma directa (Puedo decir esto porque tengo fotos de él y de cierto defensor de PowerPC en una posición comprometida).

4.10. Cronómetros de Carreras: Un Pasatiempo del Núcleo

Los cronómetros pueden producir sus propios problemas con las carreras. Considera una colección de objeros (listas, hash, etc) donde cada objeto tiene un cronómetro que lo va a destruir.

Si quieres destruir la colección entera (digamos en el borrado de un módulo), quizás realices lo siguiente:

```
/* ESTE CÓDIGO ES MALO MALO MALO MALO: SI HUBIERA ALGO PEOR
   USUARÍA NOTACIÓN HÚNGARA */
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);
```

Tarde o temprano, esto rompería en SMP, porque un cronómetro puede acabar antes que `spin_lock_bh()`, y sólo obtendría el bloqueo después de `spin_unlock_bh()`, y entonces intentaría liberar el elemento (¡el cual ya ha sido liberado!).

Esto puede ser eliminado comprobando el resultado de `del_timer()`: si retorna 1, el cronómetro ha sido borrado. Si 0, significa (en este caso) que está actualmente ejecutándose, por lo tanto lo que podemos hacer es:

```
retry:
        spin_lock_bh(&list_lock);
```

```
while (list) {
        struct foo *next = list->next;
        if (!del_timer(&list->timer)) {
                /* Le da al cronómetro una oportunidad para borrarlo */
                spin_unlock_bh(&list_lock);
                goto retry;
        }
        kfree(list);
        list = next;
}

spin_unlock_bh(&list_lock);
```

Otro problema común es el borrando de cronómetros que se reinician a ellos mismos (llamando a `add_timer()` al final de su función cronómetro). Porque este es un caso bastante común que es propenso a carreras, puedes poner una llamada a `timer_exit()` muy al funal de tu función cronómetro, y usar `del_timer_sync()` para manejar este caso. Él retorna el número de veces que el cronómetro tuvo que ser borrado antes de que finalmente lo paráramos añadiéndolo otra vez.

Capítulo 5. Lecturas Adicionales

- `Documentation/spinlocks.txt`: El tutorial de spinlocks de Linus Torvalds en los códigos del núcleo.
- Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers:

  Muy buena introdución de Curt Schimel al nivel de bloqueo de núcleo (no escrito para Linux, pero cercanamente a todo lo aplicado). El libro es caro, pero realmente vale cada penique para entender el bloqueo en SMP. [ISBN: 0201633388]

Capítulo 6. Gracias

Gracias a Telsa Gwynne por darle el formato DocBook, ordenando y añadiéndole estilo.

Gracias a Martin Pool, Philipp Rumpf, Stephen Rothwell, Paul Mackerras, Ruedi Aschwanden, Alan Cox, Manfred Spraul y Tim Waugh por la profunda lectura, corrección, encendido y comentarios.

Gracias a la intriga por no tener influencia en este documento.

Capítulo 7. Sobre la Traducción

Este documento es la traducción de "Unreliable Guide To Locking", documento que acompaña al código del núcleo de Linux, versión 2.4.18.

Este documento ha sido traducido por Rubén Melcón <melkon@terra.es>; y es publicado por el Proyecto Lucas (http://lucas.hispalinux.es)

Versión de la tradución 0.04 (Julio de 2002).

Si tienes comentarios sobre la traducción, ponte en contacto con Rubén Melcón <melkon@terra.es>

Glosario

bh

Bottom Half: por motivos históricos, las funciones con '_bh' en ellas frecuentemente ahora se refieren a cualquier interrupción software, ej. `spin_lock_bh()` bloquea cualquier interrupción software en la CPU actual. Los Bottom Halves están desaprobados, y serán eventualmente reemplazados por las tasklets. Sólo un bottom half se estará ejecutando a la vez.

Interrupción Hardware / IRQ Hardware

Petición de interrupción Hardware. `in_irq()` retorna true en un manejador de interrupciones hardware (también retorna true cuando las interrupciones son bloqueadas).

Contexto de Interrupciones

No el contexto de usuario: procesando una irq hardware o software. Indicado por la macro `in_interrupt()` retornando true (aunque también retorna true cuando las interrupciones o los BHs son bloqueados).

SMP

Symmetric Multi-Processor (Multi-Procesamiento Simétrico): núcleos compilados para máquinas con múltiples CPUs. (CONFIG_SMP=y).

softirq

Estrictamente hablando, una de las 32 interuupciones software enumeradas que pueden ejecutarse en múltiples CPUs a la vez. Algunas veces usadas también para referirse a las tasklets y bottom halves (esto es, todas las interrupciones software).

Interrupción Software / IRQ Software

Manejador de interrupciones software. `in_irq()` retorna false; `in_softirq()` retorna true. Tasklets, softirqs y bottom halves caen todos en la categoría de 'interrupciones software'.

tasklet

Una interrupción software dinámicamente registrable, que está garantizada que sólo se ejecutará en una CPU a la vez.

UP

Uni-Processor (Mono-Procesador): No-SMP. (CONFIG_SMP=n).

Contexto de Usuario

El núcleo ejecutándose en nombre de un proceso particular o hilo del núcleo (dado por la macro `current()`). No te confundas con el espacio de usuario. Puede ser interrumpido por las interrupciones software o hardware.

Espacio de Usuario

Un proceso ejecutando su propio código fuera del núcleo.