Concurrencia y mecanismos de sincronización en el kernel Linux

Matías Zabaljáuregui

Contenidos

0. Introducción

Todos los kernels Unix modernos son reentrantes, es decir, es posible que más de un proceso pueda estar ejecutándose a la vez en modo Kernel. En el informe se estudiarán y compararán los mecanismos de sincronización ofrecidos por el kernel Linux corriendo sobre una arquitectura x86, explicando cómo pueden combinarse, agregando ejemplos reales obtenidos del código y mencionando los cambios introducidos en la última versión del kernel. Previamente se introducen algunos conceptos importantes que serán usados en las explicaciones posteriores.

Este informe puede interpretarse como un estudio de algunos de los temas mencionados en el capítulo 6 del libro de Gregory R. Andrews, “Multithreaded, Parallel, and Distributed Programming”, aplicados al caso real del kernel del sistema operativo GNU/Linux. Sin embargo debe notarse que en dicho libro se estudian las posibles implementaciones de mecanismos de sincronización ofrecidos a los procesos ejecutándose en modo usuario y en éste documento se analizan las formas de sincronización de procesos en modo kernel. Ésta diferencia puede observarse más claramente si vemos que las primitivas mencionadas en el libro se invocan a través de llamadas al sistema mientras que aquí las operaciones son simples funciones a las cuales los procesos en modo kernel acceden directamente.

Cabe aclarar que se tomó la decisión de no traducir ciertas palabras pertenecientes a la terminología utilizada normalmente en estudios de sistemas operativos. A mi entender, cualquier intento por reemplazar palabras como “preemptive” o “bottom halves” por sus traducciones literales agrega un nivel de confusión al texto ya que éstos son términos universalmente conocidos en idioma Inglés.

1. Conceptos Previos

1.1 Modo kernel vs modo usuario

En términos generales una CPU tiene dos modos de funcionamiento, modo usuario y modo kernel o privilegiado. Los procesadores x86 tienen cuatro estados, pero los kernels Unix usan sólo los mencionados. Un proceso normalmente se ejecuta en modo usuario y cambia a modo privilegiado sólo cuando requiere un servicio ofrecido por el kernel, ya que en este modo puede acceder directamente a las estructuras de datos y rutinas del kernel. Cada modelo de CPU ofrece instrucciones especiales para cambiar su modo de funcionamiento y esto ocurrirá como consecuencia de alguno de los siguientes eventos:

- Un proceso invoca una llamada al sistema.
- La CPU eleva una excepción.
- Un dispositivo activa una señal de interrupción.

1.2 Kernel reentrante

Linux es un kernel reentrante. Esto significa que mas de un proceso puede estar ejecutándose en modo kernel al mismo tiempo. Por supuesto, en sistemas monoprocesador, sólo un proceso puede progresar, pero el resto puede estar bloqueado en modo kernel esperando por la CPU o que se complete alguna operación de E/S. Si ocurre una interrupción de hardware (explicada más abajo), un kernel reentrante es capaz de suspender el proceso actual, incluso si este proceso esta ejecutándose en modo kernel. Esta capacidad es muy importante, ya que mejora el rendimiento de los controladores de dispositivos (pues una vez que un dispositivo ha activado su señal de interrupción, se queda esperando a que la CPU la confirme).

1.3 Interrupciones y excepciones

Una interrupción suele ser definida como un evento que altera la secuencia de instrucciones ejecutadas por un procesador. Esos eventos corresponden a señales eléctricas generadas por hardware desde dentro y fuera de la CPU. Suelen clasificarse en:

- Excepciones: producidas por la unidad de control de la CPU.
- Interrupciones: generadas por otros dispositivos de hardware.

Cuando una señal de interrupción llega a la CPU, el kernel debe realizar ciertas tareas que generalmente son implementadas desde un módulo llamado manejador de interrupción. No todas las acciones a ser realizadas tienen la misma urgencia. De hecho, el manejador de interrupciones no es un entorno en el que puedan realizarse cualquier tipo de acción. Las operaciones largas y no críticas deberían ser diferidas ya que mientras el manejador se está ejecutando, las señales en la línea IRQ correspondiente son temporalmente ignoradas. Más importante aún es el hecho de que el proceso ejecutándose en el momento en que el manejador se dispara siempre debe estar en estado RUNNING o el sistema puede congelarse. Por lo tanto, los manejadores de interrupciones no pueden realizar operaciones bloqueantes (aquellas que pueden suspender al proceso actual o "current").

Linux divide las acciones a ser realizadas por una interrupción en tres clases:

- Críticas: Acciones como confirmar la interrupción al PIC (Controlador Programable de Interrupciones), reprogramar al dispositivo de hardware, acceder a estructuras de datos compartidas por el dispositivo y el procesador, etc. Estas son ejecutadas rápidamente, desde el manejador y con las interrupciones enmascarables deshabilitadas.

- No Críticas: Acciones como leer a datos accedidos sólo desde el procesador (por ejemplo, leer el "scan code" despues que una tecla fue presionada). Se ejecutan desde el manejador con las interrupciones habilitadas.

- Diferibles: Acciones como copiar el contenido de un buffer en el espacio de algún proceso. Pueden ser postergadas por un intervalo de tiempo sin afectar las operaciones del kernel. Son realizadas desde funciones separadas explicadas a continuación.

1.4 Funciones Diferibles: "Softirqs", "Tasklets", y "Bottom Halves"

Debe recordarse que las rutinas de servicio de un manejador de interrupciones son serializadas; no debería haber ocurrencias de una interrupción determinada hasta que el correspondiente manejador haya terminado. Por otro lado las tareas diferibles mencionadas anteriormente se ejecutan con todas las interrupciones habilitadas y sacarlas del manejador ayuda a mantener reducido el tiempo de respuesta del kernel.

Linux implementa esto usando tres tipos de funciones diferibles del kernel: "softirqs", "tasklets" y "bottom halves". Aunque estos tipos de funciones se implementan de manera diferente, están muy relacionadas. De hecho, el término "softirq", que aparece en el código fuente, usualmente denota todos los tipos de funciones diferibles y en la sección 3 del informe, éste término se usará de esa forma.

Como regla general, una softirq no puede ser interrumpida para ejecutar otra softirq en la misma CPU. Lo mismo sucede con las otras funciones diferibles. Sin embargo, en un sistema multiprocesador, varias funciones diferibles pueden ejecutarse concurrentemente en diferentes CPUs. El grado de concurrencia depende del tipo de función:

- Las softirq son reentrantes, es decir, pueden ejecutarse en varias CPUs concurrentemente, incluso si son del mismo tipo.
- Las tasklets de diferentes tipos pueden ejecutarse concurrentemente, pero las del mismo tipo no.
- Las bottom halves son globalmente serializadas; cuando una se está ejecutando en alguna CPU, ninguna otra lo estará haciendo en otra CPU. Esta es una gran limitación de Linux para sistemas multiprocesador, y de hecho, las bottom halves sólo se mantienen por cuestiones de compatibilidad. (Se espera que los desarrolladores de drivers se actualicen reemplazando las bottom halves por tasklets, y que las bottom halves desaparezcan del kernel.)

Éste tipo de funciones suele ser planificada por el programador desde un manejador de interrupciones (o de otro contexto) para que se ejecute asincrónicamente en algún momento “seguro” (elegido por el kernel) en el futuro cercano

En el kernel 2.6 se introducen las workqueues, que si bien representan una forma de planificar una función para ser ejecutada en el futuro, tienen diferencias importantes con las funciones diferibles mencionadas.

1.5 Caminos de control del kernel

Un camino de control del kernel (Kernel Control Path, KCP a partir de ahora) denota una secuencia de instrucciones ejecutada por el kernel para manejar una llamada al sistema, una excepción o una interrupción. En el caso más simple, la CPU ejecuta el KCP secuencialmente, desde la primer instrucción hasta la última. Sin embargo, cuando uno de los siguientes eventos ocurre, la CPU intercala KCPs:

- Un proceso ejecutándose en modo usuario invoca una llamada al sistema, y el KCP correspondiente verifica que el requerimiento realizado no puede satisfacerse inmediatamente. Entonces invoca al scheduler para elegir un nuevo proceso. Se realiza el context switch y un nuevo proceso comienza a ejecutarse. Éste podria hacer una llamada al sistema y tendríamos 2 procesos en modo kernel.

- La CPU detecta una excepción (por ejemplo, el acceso a una página no presente en RAM) mientras está ejecutando un KCP. Éste es suspendido y la CPU comienza la ejecución del manejador de la excepción.

- Una interrupción de hardware ocurre mientras la CPU se encuentra ejecutando un KCP (con las interrupciones activadas). Similar al anterior.

- Una función diferible comienza a ejecutarse, mientras la CPU se encuentra ejecutando un KCP.

La Implementación de un kernel reentrante requiere el uso de mecanismos de sincronización. Si un KCP es suspendido mientras modifica una estructura de datos del kernel, ningún KCP debería poder actuar sobre la misma estructura hasta que se alcance un estado consistente.

1.6 Condición de Preemptive

Linux, en sus versiones hasta la 2.4 fue un kernel nonpreemptive, al igual que los kernels Unix más tradicionales. En particular, en las versiones indicadas de Linux se cumplen las siguientes afirmaciones:

- Cuando un proceso se ejecuta en Modo Kernel, no puede ser arbitrariamente suspendido y sustituido por otro proceso, excepto cuando el primero voluntariamente delega el control de la CPU.
- Los manejadores de interrupciones, excepciones y funciones diferibles pueden interrumpir un proceso corriendo en Modo Kernel, sin embargo, cuando el manejador termina, el KCP del proceso original continua su ejecución (no se invoca al scheduler para elegir a otro proceso).

- Un KCP ejecutando un manejador de interrupciones no puede ser interrumpido por un KCP ejecutando una función diferible o una rutina de servicio a una llamada del sistema.

Gracias a estas condiciones, en un sistema monoprocesador, los KCP que implementan una llamada al sistema no bloqueante son atómicos con respecto a otros KCP iniciados por llamadas al sistema. Esto simplifica la implementación de muchas funciones del kernel: cualquier estructura de datos que no sea modificada por manejadores de interrupciones, excepciones o funciones diferibles pueden ser accedidas de manera segura. Sin embargo si un proceso en Modo Kernel voluntariamente cede la CPU, debe asegurarse de dejar todas las estructuras de datos en un estado consistente.

En sistemas multiprocesador, muchas CPUs pueden ejecutar código del kernel al mismo tiempo, por lo tanto no puede asumirse que una estructura pueda ser accedida de manera segura sólo porque no es modificada por un manejador.

Desde la rama 2.4, existen varias propuestas para disminuir la latencia del scheduler de Linux. Dos grandes grupos de parches, llamados “preemption patchs” y “low-latency patchs” son soluciones alternativas a este problema que siguen estrategias diferentes. El primer grupo introduce características de preemptive al kernel linux.

Una de las mejoras definitivas en Linux 2.6 es que el kernel es por fin preemptible, por lo que deben tomarse las precauciones necesarias para evitar interferencia, incluso en sistemas monoprocesador.

2. Colas de Espera (Wait Queues)

Inicialmente se consideró esta sección como parte de los “Conceptos Previos”. Sin embargo dada la importancia de estos mecanismos y la analogía que presentan con las “condition variables” y las “colas de espera” presentadas en el libro de Andrews, decidí crear una sección exclusiva para su descrición.

2.1 Dormir y despertar a un proceso

Cuando un proceso debe esperar por un evento (como el arribo de datos de un dispositivo), éste debería dormirse, lo que significa que el proceso suspende su ejecución y libera al procesador para otros usos. Cuando ocurra el evento esperado, el proceso será despertado y continuará con su trabajo.

Existen muchos métodos para dormir y despertar procesos, cada uno pensado para una situación distinta. Sin embargo, todos trabajan con el mismo tipo de datos básico: una cola de espera (implementada por el tipo wait_queue_head_t). Ésta representa una cola de procesos que esperan por un evento.

Se declaran e inicializan de esta forma:

```
wait_queue_head_t miColaEspera;

init_waitqueue_head (&miColaEspera);
```

O estáticamente con esta macro, que también la inicializa:

```
DECLARE_WAIT_QUEUE_HEAD (miColaEspera);
```

Las funciones para dormir a un proceso en una cola son varias y su uso depende de cuan profundo queremos que el proceso se duerma:

- sleep_on(wait_queue_head_t *miColaEspera);

Pone al proceso que la invoca a dormir en miColaEspera. Tiene la desventaja de no ser interruptible, por lo tanto el proceso podría bloquearse (siendo imposible matarlo) si el evento por el cual espera nunca ocurre.

- interruptible_sleep_on(wait_queue_head_t *miColaEspera);

La versión interruptible permite que el proceso reciba una señal de interrupción.

- sleep_on_timeout(wait_queue_head_t *miColaEspera, long timeout);
- interruptible_sleep_on_timeout(wait_queue_head_t *miColaEspera, long timeout);

Éstas variantes agregan un tiempo máximo (timeout), luego del cual el proceso será despertado aunque no haya ocurrido el evento esperado.

- void wait_event(wait_queue_head_t miColaEspera, int condicion);
- int wait_event_interruptible(wait_queue_head_t miColaEspera, int condicion);

Estas macros representan el método recomendado para dormir a un proceso. Éste dormirá hasta que la condición (puede ser cualquier expresión booleana en C) se evalúe en verdadero.

Así como hay varias formas de dormir a un proceso, existen varias formas de despertarlo:

- wake_up(wait_queue_head_t *miColaEspera);

Esta función despertará a todos los procesos que esperan en miColaEspera.

- wake_up_interruptible(wait_queue_head_t *miColaEspera);

Solo despierta a los procesos que fueron dormidos con la opción interrumpible.

- wake_up_sync(wait_queue_head_t *miColaEspera);
- wake_up_interruptible_sync(wait_queue_head_t *miColaEspera);

Normalmente una llamada wake_up puede causar un reschedule inmediato, lo que significa que otro proceso podría correr antes que la llamada a wake_up retorne. En cambio, las variantes "sincrónicas" ponen a todos los procesos despertados en estado RUNNABLE, pero no hacen un reschedule. Debe notarse que los procesos despertados podrían correr inmediatamente en otro procesador, por lo tanto no debe esperarse que estas funciones provean exclusión mutua.

2.2 Esperas exclusivas

El tipo wait_queue_head_t es una estructura sencilla, definida en <linux/wait.h>. Sólo contiene un lock y una lista enlazada de procesos que duermen. Cada nodo de la lista es del tipo wait_queue_t, y suele ser alocado (como variable automática) en funciones como interruptible_sleep_on, por lo que el programador no debe manipularlo directamente.

Sin embargo, algunas situaciones requieren el uso directo de este tipo de variables por parte del programador. El siguiente código es una versión simplificada de la implementación de interruptible_sleep_on para poner un proceso a dormir:

```
void sleep_on_SIMPLIFICADA(wait_queue_head_t *queue)
{
  wait_queue_t wait;
  init_waitqueue_entry(&wait, current);
  current->state = TASK_INTERRUPTIBLE;
  add_wait_queue(queue, &wait);
  schedule();
  remove_wait_queue (queue, &wait);
 }
```

Primero se crea la variable wait y se inicializa. También se setea el estado del proceso a TASK_INTERRUPTIBLE, lo que significa que dormirá pero puede ser despertado por una señal. Luego se agrega wait a la cola de espera queue (recibida como parámetro) y se invoca al scheduler, cediendo el procesador a otro proceso. La función schedule() retornará sólo cuando otro proceso haya despertado al actual y haya seteado su estado a TASK_RUNNING. Por último, se remueve el “nodo” de la cola de espera.

El programador, en principio, puede no necesitar conocer estos detalles, sin embargo existen varias razones para entender el funcionamiento de las colas de espera con un nivel más de profundidad. Una razón es la de hacer esperas exclusivas. Las funciones vistas hasta ahora despiertan a TODOS los procesos en la cola. Si sólo se desea despertar a uno, despertar a todos los procesos en la cola para que luego vuelvan a dormir genera context switches innecesarios (y por lo tanto tiempo de procesamiento perdido).

Por esta razón, la serie de desarrollo 2.3 del kernel agregó el concepto de una espera exclusiva, lo que significa que el kernel despertará a los procesos de a uno, con sucesivas llamadas a wake_up. El código (simplificado) para realizar una espera exclusiva es muy similar a la espera normal.

```
void sleep_exclusive_SIMPLIFICADA(wait_queue_head_t *queue)
{
  wait_queue_t wait;
  init_waitqueue_entry(&wait, current);
  current->state = TASK_INTERRUPTIBLE | TASK_EXCLUSIVE;
  add_wait_queue_exclusive(queue, &wait);
  schedule();
  remove_wait_queue (queue, &wait);
 }
```

Agregar el flag TASK_EXCLUSIVE al estado del proceso indica que ese proceso estará en una espera exclusiva. Además es necesario llamar a la función add_wait_queue_exclusive, la cual agrega al proceso al FINAL de la cola de espera. De esta forma, los procesos en esperas normales quedarán siempre delante de aquellos en espera exclusiva. Tan pronto como wake_up alcance el primer proceso en espera exclusiva, sabrá que debe dejar de despertar procesos.

Como nota final, en la versión 2.6 del kernel se agregó una nueva función:

- int wait_event_interruptible_timeout(queue, condition, timeout);

Termina la espera si expira el período de timeout.

3. Primitivas de sincronización

A continuación se mencionan las alternativas que existen para poder intercalar KCP evitando condiciones de carrera (race conditions). Éste es el tema central del informe. La siguiente tabla lista las técnicas de sincronización usadas por el kernel Linux. La columna "Alcance" indica si la técnica se aplica a todas las CPUs del sistema o a una CPU en particular (en el caso de sistemas multiprocesador). Debe recordarse que la palabra "softirq" se usará como sinónimo de función diferible, a no ser que se aclare otra cosa.

Técnica	Descripción	Alcance
Operación Atómica	Instrucción para leer-modificar-escribir un contador atómicamente	Todas las CPUs
Barrera de Memoria	Evita la reordenación de instrucciones	CPU local
Spinlock	Lock con "busy wait"	Todas las CPUs
Semáforo	Lock con "blocking wait" (dormir)	Todas las CPUs
Deshabilitación de interrupciones locales	Prohibe manejar interrupciones en una CPU	CPU local
Deshabilitación de softirq locales	Prohibe la ejecución de softirq en una CPU	CPU local
Deshabilitación de interrupciones globales	Prohibe interrupciones y softirq en todas las CPUs	Todas las CPUs

A continuación se explican con más detalle las técnicas de sincronización. Luego se estudian las formas en que pueden combinarse para conseguir los mejores resultados.

3.1 Operaciones Atómicas

Existen varias instrucciones de lenguaje ensamblador del tipo leer-modificar-escribir; acceden a una posición de memoria dos veces, la primera para leer el valor antiguo y la segunda para escribir el nuevo. Estas operaciones deben ser ejecutadas en una única instrucción sin ser interrumpidas y evitando accesos a la mismo posición de memoria por otras CPUs. En el libro de Andrews (en el Capítulo 3) se ofrece una descripción de éste tipo de instrucciones de bajo nivel, poniendo como ejemplos a Test-and-Set y Fetch-and-Add.

Los procesadores 80 x 86 nos ofrecen las siguientes alternativas:

- Instrucciones del lenguaje ensamblador que hacen cero o un acceso alineado a memoria. Un dato esta alineado en memoria cuando su dirección es un múltiplo de su tamaño en

bytes. Por ejemplo, la dirección de un entero alineado debe ser múltiplo de dos (en 80x86, un dato del tipo int ocupa 16 bits). En términos generales, un acceso a memoria no alineado es no atómico.

- Las instrucciones leer-modificar-escribir como inc o dec son atómicas si otro procesador no toma el bus despues de la lectura o antes de la escritura. El robo del bus de memoria nunca sucede en un sistema monoprocesador.
- Las instrucciones leer-modificar-escribir cuyo código de operación es precedido por el byte lock (0xf0) son atómicas incluso en sistemas multiprocesador. Cuando la unidad de control detecta ese prefijo, bloquea el bus de memoria hasta que la instrucción termine.

El kernel Linux proveé un tipo especial **atomic_t** (un contador de 24 bits accesible atómicamente) y algunas funciones especiales (ver la siguiente tabla) que actúan sobre variables de este tipo y que están implementadas como instrucciones atómicas escritas en ensamblador. En sistemas multiprocesador, estas instrucciones son precedidas por el byte lock.

Función	Descripción
Atomic_read(v)	Retorna *v
Atomic_set(v,i)	Setea *v a i
Atomic_add(i,v)	Suma i a *v
Atomic_sub(i,v)	Resta i de *v
Atomic_sub_and_test(i, v)	Resta i de *v y retorna 1 si el resultado es cero; de lo contrario retorna 0.
Atomic_inc(v)	Suma 1 a *v
Atomic_dec(v)	Resta 1 de *v
Atomic_dec_and_test(v)	Resta 1 de *v y retorna 1 si el resultado es cero; de lo contrario retorna 0.
Atomic_inc_and_test(v)	Suma 1 a *v y retorna 1 si el resultado es cero; de lo contrario retorna 0.
Atomic_add_negative(i, v)	Suma 1 a *v y retorna 1 si el resultado es negativo; de lo contrario retorna 0.

Otra clase de funciones atomicas operan en máscaras de bits.

Función	Descripción
test_bit(n, dir)	Retorna el valor del bit n de *dir
set_bit(n, dir)	Setea el bit n de *dir
Clear_bit(n, dir)	Pone a cero el bit n de *dir
Change_bit(n, dir)	Invierte el bit n de *dir
test_and_set_bit(n, dir)	Setea el bit n de *dir y retorna su antiguo valor
test_and_clear_bit(n, dir)	Pone a 0 el bit n de *dir y retorna su antiguo valor
test_and_change_bit(n, dir)	Invierte el bit n de *dir y retorna su antiguo valor
Atomic_clear_mask(masc, dir)	Pone a 0 todos los bits de dir especificados por masc
Atomic_set_mask(masc, dir)	Setea todos los bits de dir especificados por masc

3.2 Barrerras de memoria

Cuando se utilizan compiladores con funciones de optimización, nunca debería darse por supuesto que las instrucciones serán ejecutadas en el orden exacto en el que aparecen en

el código fuente. Un compilador podría reordenar las instrucciones ensamblador para optimizar el uso de los registros, o podrían reordenarse los accesos a memoria en el caso de las CPUs superescalares.

Sin embargo, a los efectos de la sincronización, la reordenación debería evitarse. Sería un problema que una instrucción ubicada despues de una primitiva de sincronización se ejecutada antes de la primitiva misma.

Una barrera de memoria asegura que las operaciones colocadas antes de la primitiva sean terminadas antes de comenzar las operaciones colocadas despues de la primitiva.

En procesadores 80x86 los siguientes tipos de de instrucciones son llamados serializadores ya que actuan como barreras de memoria:

- Instrucciones que operan en puertos I/O
- Instrucciones precedidas por el byte lock
- Instrucciones que escriben en registros de control, registros de sistema o registros de depuración (por ejemplo, **cli** y **sti**).
- Algunas instrucciones especiales (por ejemplo **iret**)

Linux usa seis primitivas mostradas en la siguiente tabla. Las barreras de memoria de Lectura actúan sólo en instrucciones que leen de la memoria, mientras que las barreras de memoria de Escritura actúan en aquellas instrucciones que escriben en memoria.

Macro	Descripción
mb( )	Barrera de memoria para P y MP
rmb( )	Barrera de memoria de lectura para P y MP
wmb( )	Barrera de memoria de escritura para P y MP
smp_mb( )	Barrera de memoria sólo para MP
smp_rmb( )	Barrera de memoria de lectura sólo para MP
smp_wmb( )	Barrera de memoria de escritura sólo para MP

P = Monoprocesador
MP = MultiProcesador

Debe notarse que en los sistemas multiprocesadores, todas las operaciones atómicas descriptas en la sección anterior actúan como barreras de memoria porque usan el byte lock.

3.3 SpinLocks

Los spinlocks son tipos especiales de locks diseñados para trabajar en entornos multiprocesador. Si el KCP encuentra el lock abierto, lo toma y continua su ejecución. Por el contrario, si el KCP encuentra el lock cerrado, se queda ejecutando un bucle hasta que el lock es liberado. Aunque representan una “busy wait”, generalmente los spinlocks son convenientes ya que muchos recursos del kernel son bloqueados por sólo una fracción de milisegundo y consumiría más tiempo que un proceso esperando por uno de esos recursos ceda la CPU y tenga que conseguirla más tarde.

Por supuesto, los spinlocks no sirven en un sistema monoprocesador, ya que el KCP que espera seguirá ejecutandose y el KCP que mantiene el lock no tendrá oportunidad de liberarlo.

En Linux, un spinlock se representa por una estructura spinlock_t, que consiste en un único campo lock. El valor 1 corresponde a un estado liberado, mientras que cualquier valor igual o menor a cero indica un estado de cerrado.

Las cinco funciones mostradas en la siguiente tabla son utilizadas para incializar, testear y setear los spinlocks. En sistemas monoprocesador, ninguna de estas funciones hace algo, excepto spin_trylock(), que siempre devuelve 1. Todas estas funciones se basan en operaciones atomicas; esto asegura que el acceso concurrente a la estructura se hará correctamente.

Función	Descripción
spin_lock_init( )	Inicializa y setea el spinlock a 1 (no bloqueado)
spin_lock( )	Espera hasta que el spinlock sea 1, luego lo setea a 0 (bloqueado)
spin_unlock( )	Setea el spinlock a 1.
spin_unlock_wait( )	Espera hasta que el spinlock sea 1
spin_is_locked( )	Retorna 0 si el spinlock está seteado a 1; de lo contrario retorna 1.
spin_trylock( )	Si está desbloqueado, setea el spinlock a 0 y retorna 1; de lo contrario retorna 0

3.4 Spinlocks de Lectura/Escritura

Los Spinlocks Lectura/Escritura fueron introducidos para incrementar la concurrencia en el kernel. Permiten que varios KCP lean simultáneamente la misma estructura de datos, siempre que ninguno la modifique. Si un KCP desea escribir en la estructura debe tomar la versión de Escritura del lock, la cual brinda acceso exclusivo al recurso.

Cada spinlock de este tipo es una estructura rwlock_t. Su campo lock es de 32 bits y codifica dos datos:

- Un contador de 24 bits que indica la cantidad de KCP leyendo la estructura (en complemento a 2).
- Un flag “unlock” que se desactiva cuando algun KCP esta leyendo o escribiendo la estructura.

Por lo tanto el campo lock almacena el número 0x01000000 si el spin lock está liberado, el número 0x00000000 si fue tomado por un escritor y cualquier número en el rango 0x00ffffff – 0x00000001 dependiendo de la cantidad de lectores.

La macro rwlock_init inicicaliza el campo lock a 0x01000000 (desbloqueado). La macro read_lock atómicamente decremente el spin lock en 1, incrementando el número de lectores si se puede, o espera. La macro read_unlock simplemente incrementa el campo lock.

La macro write_lock intenta tomar el lock para Escritura y write_unlock lo libera.

3.5 Semaforos

Linux ofrece dos tipos de semáforos, aunque en este informe interesan sólo los primeros:

- Semáforos del kernel, usados por los KCP.
- Semáforos “System V IPC”, usados por los procesos en modo usuario.

Cuando un KCP intenta obtener un recurso ocupado protegido por un semáforo del kernel, el proceso correspondiente es suspendido. El proceso volverá al estado RUNNABLE cuando el recurso sea liberado.

Un semáforo del kernel es un objeto de tipo struct semaphore, conteniendo los campos mostrados en la siguiente lista:

count

Almacena un contador atomic_t. Si es mayor que 0, entonces el recurso está libre. Si es

igual a 0, el recurso está ocupado pero ningún otro proceso está esperando. Finalmente si count es negativo, el recurso no está disponible y al menos un proceso está esperando para usarlo.

wait

Almacena la dirección de una cola de espera (wait queue) que incluye a todos los procesos dormidos que están esperando por el recurso. Si count es mayor o igual a 0, la cola está vacía.

sleepers

Guarda un flag que indica si algún proceso está bloqueado en el semáforo.

Las macros init_MUTEX e init_MUTEX_LOCKED pueden ser utilizadas para inicializar un semaforo para acceso exclusivo: setean el campo count a 1(recurso libre) y 0 (recurso no disponible con acceso exclusivo cedido al proceso que inicializa el semaforo) respectivamente. Un semáforo podría se inicializado con cualquier valor positivo en count para permitir el acceso concurrente de más de un proceso al recurso. Cuando un proceso desea liberar un semáforo, invoca la función up( ). Por el contrario, cuando un proceso desea acceder a un recurso protegido, invoca a la función down( ).

Sólo manejadores de excepciones y algunas rutinas de servicio de llamadas al sistema pueden usar la función down( ). Los manejadores de interrupciones o funciones diferibles no deben invocar down ( ) ya que esta función suspende al proceso cuando el semáforo está ocupado. Por esta razón, linux ofrece la función down_trylock ( ) que puede ser utilizada sin problemas por las funciones asincrónicas mencionadas anteriormtente. Esta función es idéntica a down ( ) excepto que cuando el recurso está ocupado, la función retorna inmediatamente en lugar de poner al proceso a dormir.

Otra variación es down_interruptible ( ). Ésta es ampliamente utilizada por drivers ya que permite que los procesos dormidos en un semáforo reciban una señal para desbloquearse, en cuyo caso el driver podría abortar la operación I/O.

3.6 Semáforos de Lectura/Escritura

Estos semáforos son una característica aparecida en la versión 2.4 de Linux. Son similares a los spinlocks de Lectur/Escritura, excepto que los procesos que esperan son suspendidos hasta que el recurso se libera.

El kernel maneja a los procesos en un orden FIFO. Cada lector o escritor que encuentra el semáforo bloqueado es agregado en la última posición de la cola de espera del semáforo. Cuando este es liberado, el proceso en la primer posición de la wait queue list es despertado y chequeado. Si es un escritor, el resto de los procesos continúan durmiendo. Si es un lector, el resto de los lectores que le sigan en la lista serán despertados y tomarán el semáforo, hasta encontrar un escritor.

Un semáforo de lectura/escritura es modelado con una estructura rw_semaphore que incluye los siguientes campos:

count

Son dos contadores de 16 bits. El contador en la palabra más significativa almacena en complemento a 2 el número de KCPs que esperan. El otro contador almacena el número total de escritores y lectores en el recurso.

wait_list

Apunta a una lista de procesos esperando. Cada elemento en la lista es una estructura rwsem_waiter, incluyendo un puntero al descriptor del proceso que espera y un flag indicando si el proceso quiere leer o escribir.

wait_lock

Un spin lock utilizado para proteger la cola de espera y a la estructura rw_semaphore.

La función init_rwsem ( ) incializa una estructura rw_semaphore seteando el campo count a 0, el spin lock wait_lock a desbloqueado y la cola wait_list a una lista vacía. Las funciones down_read( ) y down_write ( ) toman el semáforo para lectura y escritura respectivamente; up_read ( ) y up_write ( ) liberan el semáforo de lectura/escritura.

3.7 Completions

Las "Completions" (terminaciones) fueron introducidos en la versión 2.4.7 del kernel, para solucionar una condición de carrera bastante sutil que ocurre en sistemas multiprocesador cuando se suceden los siguientes eventos:

- Un proceso A aloca un semáforo temporal y lo inicializa como un "close MUTEX", le pasa su dirección a otro proceso B y llama a down ( ) sobre éste semáforo.

- Luego, el proceso B corriendo en una CPU diferente invoca a up( ) sobre el mismo semáforo.

La implementación actual de up( ) y down( ) permite que se ejecuten en concurrentemente sobre el mismo semáforo. Por lo tanto el proceso A puede ser despertado y destruir el semáforo temporal mientras el proceso B todavía se encuentra ejecutando la función up(). Como resultado, up() podría intentar acceder a una estructura de datos que no existe más.

Una solución posible a éste problema es modificar la implementación de down() y up() para prohibir las ejecuciones concurrentes sobre el mismo semáforo. Sin embargo, éste cambio requeriría instrucciones adicionales, algo que se intenta evitar en éste tipo de funciones que son tan frecuentemente utilizadas.

Las Completions son primitivas de sincronización específicamente diseñadas para resolver este problema. La estructura de datos completion incluye una cola de espera y un flag:

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

Trabajar con completions requiere una creación e inicialización estática:

DECLARE_COMPLETION(my_comp);

O la declaración de una variable automática (o dinámica) y su inicialización:

```
struct completion my_comp;
init_completion(&my_comp);
```

La función análoga al up() de los semáforos se denomina complete(). Recibe como argumento la dirección de una estructura completion, setea el flag done a 1 e invoca a wake_up() para despertar a un proceso exclusivo durmiendo en la cola de espera. La función complete_all() despertará a todos los procesos durmiendo en la cola de espera.

La función correspondiente a down() se llama wait_for_completion(). Recibe la dirección de una estructura completion y verifica el valor del flag done. Si está seteado a 1, la función termina porque complete() ya fue ejecutada en otra CPU. En otro caso, wait_for_completion() pone a dormir al proceso current en el estado TASK_UNINTERRUPTIBLE. Una vez despertado, la función quita al proceso current de la cola de espera, setea done a 0 y termina.

La diferencia real entre las completions y los semáforos es la forma de utilizar el spin lock incluido en la cola de espera. Las funciones complete() y wait_for_completion() lo usan para asegurarse de que no puedan ejecutarse concurrentemente, mientras que up() y down() lo usan sólo para serializar accesos a la cola de espera.

3.8 Deshabilitación local de interrupciones

Deshabiltar las interrupciones es uno de los mecanismos clave utilizado para asegurar que una secuencia de sentencias del kernel es tratada como una sección crítica. Permite que un KCP continue su ejecución aun cuando un dispositivo de hardware active una señal de interrupción y de esta forma proveé una manera efectiva de proteger estructuras de datos que también son accedidas por manejadores de interrupciones. Sin embargo, en sistemas multiprocesador, la deshabilitación de interrupciones locales no proteje contra accesos concurrentes de manejadores de interrupciones que se ejecuten en otras CPUs. Es por eso que suelen combinarse con spinlocks.

Las interrupciones pueden ser deshabilitadas en una CPU x86 con la instrucción asembler cli, la cual es utilizada en la macro __cli ( ). Las interrupciones pueden ser habilitadas por sti, contenida en la macro __sti( ). Las versiones recientes del kernel también definen las macros local_irq_disable( ) y local_irq_enable( ), que son equivalentes a las mencionadas anteriormente, pero cuyos nombres son más fáciles de entender y no dependen de la arquitectura.

La instrucción asembler cli, pone a cero el flag IF del registro eflags para deshabilitar las interrupciones, pero al final de la sección crítica no se puede simplemente setear el flag nuevamente. Esto es asi porque los manejadores de interrupciones (de interrupciones diferentes) pueden ejecutarse de manera anidada, por lo tanto el kernel no conoce necesariamente cual era el valor del flag IF antes de que se ejecutara el KCP actual. En estos casos, se debe guardar el valor anterior y luego restaurarlo a su valor original.

Para esto se utilizan las macros __save_flags( ) y __restore_flags( ). Típicamente estas macros se usan de la siguiente manera:

```
_ _save_flags(valorOriginal);
 _cli(  );
[...]
_ _restore_flags(valorOriginal);
```

Entonces las interrupciones serán reactivadas sólo si estaban habilitadas antes que el actual KCP invocara a la macro __cli( ).

Las versiones recientes del kernel definen las macros local_irq_save( ) y local_irq_restore( ), que son equivalentes a las mencionadas anteriormente, pero cuyos nombres son más fáciles de entender.

3.9 Deshabilitación global de interrupciones

Algunas funciones críticas del kernel pueden ejecutarse en una CPU sólo si no se están ejecutando ningún manejador de interrupciones o función diferible en otras CPUs. Este requerimiento de sincronización se puede satisfacer deshabilitando las interrupciones globalmente. Un escenario típico consiste en un driver que necesita resetear un dispositivo de hardware. Antes de comenzar a escribir los puertos I/O, el driver deshabilita globalmente las interrupciones, asegurándose que ningún otro driver accederá a los mismos puertos durante la secuencia de reseteo.

Sin embargo, este mecanismo disminuye el nivel de concurrencia global del sistema y se está dejando de usar ya que puede ser reemplazado por técnicas de sincronización más eficientes.

La deshabilitación global de interrupciones se realiza mediante la macro cli( ). En sistemas monoprocesadores, esta macro simplemente se expande a __cli( ), deshabilitando las interrupciones locales. En sistemas multiprocesador, la macro espera hasta que terminen todos los manejadores de interrupciones y funciones diferibles, y luego toma un spin lock especial llamado global_irq_lock.

Una vez que cli( ) retorna, ningún manejador de interrupciones comenzará a ejecutarse hasta que las interrupciones sean reactivadas invocando la macro sti( ).

3.10 Deshabilitación de funciones diferibles

Como se explicó anteriormente las funciones diferibles pueden ser ejecutadas en momentos impredecibles (generalmente, al terminar los manejadores de interrupciones). Por este motivo, las estructuras de datos accedidas por las funciones diferibles deben ser protegidas contra condiciones de carrera.

Una manera trivial de impedir la ejecución de funciones diferibles en una CPU es deshabilitar las interrupciones en esa CPU. Ya que no se activarán los manejadores de interrupción, no podrán activarse las softirq para ser ejecutadas asincrónicamente. La deshabilitación global de interrupciones en todas las CPUs tambien deshabilita las funciones diferibles en todas las CPUs. Cuando se retorna de la macro cli( ), el KCP invocante puede asumir que ninguna función diferible está en ejecución y que ninguna comenzará hasta que las interrupciones sean habilitadas globalmente.

Sin embargo, a veces el kernel necesita deshabilitar las funciones diferibles sin deshabilitar las interrupciones. Las funciones diferibles pueden ser deshabilitadas en cada CPU seteando a un valor no nulo el campo __local_bh_count de la estructura irq_stat asociada con la CPU en cuestion. La macro local_bh_disable incrementa el campo __local_bh_count en 1, mientras que la macro local_bh_enable la decrementa.

4. Utilizar y combinar las primitivas de sincronización

La performance del sistema puede variar considerablemente, dependiendo del tipo de primitiva de sincronización elegida para resolver cada situación. Normalmente los desarrolladores prefieren mantener el nivel de concurrencia del kernel lo más alto posible y esto depende de dos factores principales:

1. El número de dispositivos I/O que operan concurrentemente
2. El múmero de CPUs que realizan trabajo productivo

Para maximizar el rendimiento de I/O, las interrupciones deberían ser deshabilitadas por períodos de tiempo breves, ya que en esos períodos las señales IRQs elevadas por los dispositivos serán temporalmente ignoradas, y esos dispositivos no podrán comenzar nuevas actividades. Para utilizar las CPUs eficientemente, las primitivas de sincronización basadas en "busy waiting" deberían ser evitadas cuando sea posible, ya que se desperdician ciclos del procesador.

A continuación se evaluarán el uso de spinlocks, semáforos y deshabilitación de interrupciones en diferentes situaciones, y la combinación de estas primitivas para lograr soluciones correctas. En general, la elección de alguna de las alternativas depende del tipo de KCP que accede a la estructura de datos a proteger.

Como se menciono anteriormente, la deshabilitación global de interrupciones perjudica notablemente a la performance del sistema, por lo que esta técnica de sincronización ya no se usa. De hecho, Linux 2.4 sólo la soporta por cuestiones de compatibilidad con drivers antiguos y fue removida en la versión de de desarrollo 2.5.

4.1 Proteger estructuras de datos accedidas por manejadores de excepciones

Cuando una estructura de datos es accedida sólo por un manejador de excepciones, las condiciones de carrera son normalmente fáciles de entender y prevenir. Las excepciones más comunes que plantean problemas de sincronización son las rutinas de servicio de llamadas al sistema, en las cuales la CPU opera en modo kernel para ofrecer un servicio a un programa en modo usuario. Por lo tanto, una estructura de datos accedida sólo por una excepción usualmente representa un recurso que puede ser asignada a uno o más procesos simultáneamente.

En estos casos (tanto en entornos monoprocesador como multiprocesador) se utilizan semáforos porque permiten que el proceso se suspenda hasta que el recurso vuelva a estar disponible.

4.2 Proteger estructuras de datos accedidas por manejadores de interrupciones

Si una estructura de datos es accedida sólo por el "top half" de un manejador de interrupciones, no requiere ninguna primitiva de sincronización ya que, como se explicó, un manejador no puede ejecutarse concurrentemente con respecto a sí mismo.

Sin embargo, si la estructura es accedida por varios manejadores esto no es cierto. Un manejador puede interrumpir a otro, y diferentes manejadores pueden ejecutarse concurrentemente en sistemas multiprocesador.

En sistemas monoprocesador, las condiciones de carrera deben evitarse sólo deshabilitando las interrupciones en todas las secciones críticas del manejador de interrupciones.

Debe tenerse en cuenta que un semáforo puede bloquear el proceso, por lo que no puede ser usado en un manejador de interrupciones. Un spinlock, por otro lado, puede colgar el sistema: si el manejador que está accediendo al dato es interrumpido por otro que intenta accederlo, no podrá liberar el lock, y el nuevo manejador podría quedarse ejecutando el loop indefinidamente esperando por el mismo.

En sistemas multiprocesador, el método más conveniente es deshabilitar las interrupciones locales (así los otros manejadores no interferirán) y tomar un spinlock o un spinlock de lectura/escritura que proteja la estructura de datos. Debe notarse que este spinlock no puede congelar el sistema porque incluso si un manejador de interrupciones ejecutándose en otra CPU encuentra el lock cerrado, eventualmente el manejador que lo tomó lo liberará.

El kernel linux utiliza varias macros que combinan la habilitación/deshabilitación local de interrupciones con la manipulación de spin locks. En sistemas monoprocesador estas macros sólo habilitan o deshabilitan las interrupciones locales.

Función	Descripción
spin_lock_irq(l)	local_irq_disable( ); spin_lock(l)
spin_unlock_irq(l)	spin_unlock(l); local_irq_enable( )
spin_lock_irqsave(l,f)	local_irq_save(f); spin_lock(l)
spin_unlock_irqrestore(l,f)	spin_unlock(l); local_irq_restore(f)
read_lock_irq(l)	local_irq_disable( ); read_lock(l)
read_unlock_irq(l)	read_unlock(l); local_irq_enable( )
write_lock_irq(l)	local_irq_disable( ); write_lock(l)
write_unlock_irq(l)	write_unlock(l); local_irq_enable( )
read_lock_irqsave(l,f)	local_irq_save(f); read_lock(l)
read_unlock_irqrestore(l,f)	read_unlock(l); local_irq_restore(f)
write_lock_irqsave(l,f)	local_irq_save(f); write_lock(l)
write_unlock_irqrestore(l,f)	write_unlock(l); local_irq_restore(f)

4.3 Proteger una estructura de datos accedida por funciones diferibles

El mecanismo utilizado en este caso depende mayormente del tipo de función diferible. Como se explicó, las softirqs, tasklets y bottom halves escencialmente difieren en el grado de concurrencia con el que pueden ser ejecutados. Aqui se usa el término softirq para indicar un tipo de función diferible en particular.

En sistemas monoprocesador no existen problemas de sincronización. Esto es asi porque en una CPU la ejecución de funciones diferibles es serializada, esto es una funcion diferible no puede ser interrumpida por otra. Por el contrario, en sistemas multiprocesador, más de una función diferible puede ejecutarse a la vez.

- Una estructura de datos accedida por una softirq debe ser protegida siempre, usualmente mediante un spinlock, porque la misma softirq puede ejecutarse simultáneamente en dos o más CPUs.
- Una estructura de datos accedida por sólo un tipo de tasklet no necesita protección, ya que las tasklets del mismo tipo no pueden ejecutarse concurrentemente. Sin embargo si la estructura es accedida por varios tipos de tasklets, entonces sí debe protegerse.
- Finalmente, las estructuras accedidas por bottom halves no necesitan protejerse porque éstos nunca se ejecutan concurrentemente.

4.4 Proteger una estructura de datos accedida por excepciones e interrupciones.

Ahora se considerará un dato accedido por excepciones (por ejemplo, rutinas de servicio de llamadas al sistema) y por manejadores de interrupciones.

En sistemas monoprocesador, la prevención de condiciones de carrera es bastante simple, ya que los manejadores de interrupción no son reentrantes y no pueden ser

interrumpidos por las excepciones: bastará con que el kernel acceda a la estructura con las interrupciones locales deshabilitadas para evitar problemas.

En sistemas multiprocesador, debe tenerse en cuenta la posibilidad de ejecución concurrente de excepciones e interrupciones en otras CPUs. Pueden combinarse la deshabilitación local de interrupciones con spinlocks.

A veces, puede ser preferible reemplazar el spinlock con un semáforo. Sin embargo ya que los manejadores de interrupciones no pueden ser suspendidos, deben tomar el semáforo usando un bucle y la función down_trylock ( ) (similar a como funciona un spinlock). Por otro lado las rutinas de servicios de las llamadas al sistema si pueden suspender a los procesos invocantes cuando el recurso no está disponible. Éste es el comportamiento esperado para la mayoría de las llamadas al sistema.

4.5 Proteger una estructura de datos accedida por excepciones y funciones diferibles

Este caso puede ser tratado como el anterior. De hecho, las funciones diferibles son generalmente activadas por los manejadores de interrupciones y no pueden ejecutarse manejadores de excepciones cuando se está ejecutando una función diferible. Combinar un spin lock con la deshabilitación local de interrupciones es más que suficiente. Aunque generalmente se prefiere deshabilitar sólo las funciones diferibles utilizando la macro local_bh_disable( ), permitiendo que las interrupciones sigan siendo servidas.

En sistemas multiprocesador, ser requiere el uso de spinlocks.

4.6 Proteger una estructura de datos accedida por interrupciones y funciones diferibles

Este caso es similar a aquel donde el dato es accedido por manejadores de interrupciones y excepciones. Una interrupción puede ser activada mientras se está ejecutando una función diferible, pero nunca sucederá lo opuesto. Por lo tanto las condiciones de carrera pueden evitarse desactivando las interrupciones locales.

De nuevo, en sistemas multiprocesador, se requiere un spin lock.

4.7 Proteger una estructura de datos accedida por excepciones, interrupciones y funciones diferibles

Como en los casos anteriores, la combinación de un spinlock y la deshabilitación de las interrupciones locales proveé la sincronización. No es necesario deshabilitar explícitamente las funciones diferibles porque éstas son normalmente activadas al final de un manejador de interrupciones.

5. Ejemplos de uso

Evitar problemas de sincronización siempre ha sido una tarea compleja para los desarrolladores porque requiere una completa comprensión de cómo interactúan los distintos subsistemas del kernel Linux. Para dar una idea de cómo se integran los mecanismos mencionados se mencionan algunos ejemplos concretos.

5.1 Contadores de Referencia

Los contadores de referecias son ampliamente usados dentro del kernel para evitar condiciones de carrera debido a alocaciones y liberaciones concurrentes de un recurso. Un contador de referencia es simplemente un contador de tipo atomic_t asociado con un recurso específico como una página de memoria, un módulo o un archivo. El contador es atómicamente

incrementado cuando un KCP comienza a usar el recurso y es decrementado cuando termina de usarlo. Cuando el contador llega a cero el recurso no está siendo usado, y puede ser liberado.

El sistema de archivos virtuales (VFS) de Linux proveé una capa de software que maneja todas las llamadas al sistema relacionadas a un filesytem Unix estandar. Entre sus numerosas estructuras de datos se encuentra el objeto file, el cual describe la interacción de un proceso con un archivo. Uno de sus campos es f_count (de tipo atomic_t) el cual contabiliza las referencias a la estructura. De esta forma, cuando count llega a tener un valor nulo, puede liberarse la estructura file correspondiente (se llama a la función release( ) sobre ese file).

En el kernel 2.6, los kobjects son estructuras que pueden embeberse en cualquier objeto del kernel y entre otras cosas proveen la función de contadores de referencia para ese objeto.

5.2 Semáforo del Inodo

Siguiendo con las estructuras del VFS, Linux almacena la información sobre un archivo en disco en un objeto llamado inode. Esta estructura incluye su propio semáforo en el campo i_sem.

Un gran número de condiciones de carrera pueden ocurrir durante la administración de un filesystem. Un archivo en disco es un recurso común para todos los usuarios, ya que todos los procesos pueden (potencialmente) acceder al contenido del archivo, cambiar su nombre o ubicación, borrarlo, copiarlo, etc. Todas las operaciones sobre discos son bloqueantes, por lo tanto incluso en sistemas monoprocesador, más de un proceso puede intentar acceder al archivo/directorio al mismo tiempo. Todas las posibles condiciones de carrera son protegidas por el semáforo incluido en la estructura inode.

En cualquier programa que use dos o más semáforos, existe la posibilidad de deadlock. En general Linux tiene pocos problemas de deadlock relacionados con semáforos ya que cada KCP usualmente necesita tomar sólo un semáforo a la vez. Sin embargo, los inodos presentan una de las excepciones a ésta afirmación. Por ejemplo, cada una de las rutinas de servicio de las llamadas al sistema rmdir( ) y rename( ) deben acceder a dos inodos diferentes y por lo tanto tomar los dos semáforos. Para evitar problemas de deadlocks, los requerimientos de semáforos son realizados en el orden en que las estructuras semaphore aparecen en memoria. Aquella estructura semaphore que tenga la dirección de memoria más baja será la primera en ser tomada. Ésto es una implementación de las denominadas técnicas basadas en grafos para evitar deadlocks, donde se impone un orden parcial sobre los datos a ser accedidos.

5.3 Lock global del kernel

En versiones antiguas del kernel Linux, cuando se comenzó a dar soporte SMP, se usó un lock global (conocido como **big kernel lock**, BKL). En la versión 2.0, este lock era un spin lock normal que aseguraba que sólo un procesador pudiera ejecutarse en modo kernel (parecido al primer kernel propuesto por Andrews en el capítulo 6 de su libro). El kernel 2.2 era considerablemente más flexible y dejó de depender de un solo spinlock. En su lugar, un gran número de estructuras de datos del kernel fueron protegidas por spinlocks especializados. Por otro lado el BKL siguió estando presente porque su reemplazo por pequeños locks no era una tarea trivial (debieron evitarse cuidadosemente la aparición de deadlocks y condiciones de carrera) y varias partes no relacionadas del kernel aún dependían de él.

La versión 2.4 del kernel reduce aún más el rol del BKL. Éste es principalmente usado en accesos al Sistema de Archivos Virtual y al cargar y descargar módulos del kernel.

Actualmente, mucha gente trabaja para remover completamente el BKL del kernel linux 2.6, o modificar los algoritmos para que no se necesiten los bloqueos (por ejemplo:

http://kerneltrap.org/node/3843).

Las funciones lock_kernel( ) y unlock_kernel( ) son usadas para tomar y liberar el BKL.

6. Kernel 2.6

Esta sección presenta mecanismos de sincronización introducidos en ésta nueva versión. Debería tenerse en cuenta que hubo grandes cambios concepuales y de diseño durante el desarrollo del kernel 2.6, y que aquí solo se mencionan dos mecanismos mecanismos individuales para facilitar la solución a ciertos problemas específicos. Al final de la sección, se menciona la nueva propiedad de preemptible, la cual acerca un poco más al kernel linux a convertirse en una solución viable para aplicaciones en tiempo real, aunque Linux no puede ser considerado todavía un sistema operativo de tiempo real.

6.1 Exclusión mutua con seqlocks

El kernel 2.5.60 agregó un nuevo tipo de lock llamado seqlock. Fueron creados para ser usados en situaciones donde se desea protejer un dato "pequeño", simple y accedido con frecuencia, y además existe la condición importante de que los escritores no deben sufrir de inanición.

Originalmente fueron designados para controlar el acceso a las variables que almacenan un valor de tiempo del sistema: jiffies_64 y xtime. Estas variables son leidas constantemente, por lo que su lectura debe ser una operación rápida. Pero tambien es importante que su actualización, la cual sucede ante la interrupción del "timer", no tenga que esperar a que los lectores las liberen.

Los seqlocks consisten en un spinlock común asociado a un valor entero que indica un "número de secuencia". Pueden ser declarados e inicializados de usando dos métodos alternativos:

```
#include <linux/seqlock.h>
seqlock_t lock1 = SEQLOCK_UNLOCKED;
seqlock_t lock2;
seqlock_init(&lock2);
```

Los escritores deben tener un acceso exclusivo antes de hacer cambios al dato protegido. Normalmente, se usa la siguiente secuencia de sentencias:

```
seqlock_t miSeqlock = SEQLOCK_UNLOCKED;
/* ... */
write_seqlock(&miSeqlock);
/* Realizar las modificaciones */
write_sequnlock(&miSeqlock);
```

La llamada a write_seqlock() bloquea el spinlock e incrementa el número de secuencia. Luego, write_sequnlock() incrementa nuevamente el numero de secuencia y libera el spinlock

El acceso de lectura no usa bloqueos. En su lugar, el lector usa el número de secuencia para detectar una colisión de acceso con un escritor, y reintenta la lectura si es necesario. El código utilizado para accederlo podría ser:

```
unsigned int numeroSecuencia;
do {
   numeroSecuencia = read_seqbegin(&miSeqlock);
      /* Hacer una copia del dato buscado */
   } while read_seqretry(&miSeqlock, numeroSecuencia);
```

Si el valor inicial del número de secuencia obtenido por read_seqbegin() es impar, un escritor se encontraba actualizando el dato cuando el lector comenzó a leerlo. Si éste número de secuencia inicial no coincide con el número de secuencia leido por read_seqretry() entonces el escritor apareció en la mitad del proceso de lectura. En ambos casos, el dato leido puede llegar a ser inconsistente y el lector debe intentarlo nuevamente. Sin embargo, en la mayoría de los casos, no ocurrirá una colisión de acceso y el lector accederá rápidamente al dato.

Junto con las primitivas mencionadas, se incluyen las variantes usuales para la sincronización con interrupciones locales y bottom halves:

```
void write_seqlock(seqlock_t *miSeqlock);
void write_sequnlock(seqlock_t *miSeqlock);
int write_tryseqlock(seqlock_t *miSeqlock);
void write_seqlock_irqsave(seqlock_t *miSeqlock, long flags);
void write_sequnlock_irqrestore(seqlock_t *miSeqlock, long flags);
void write_seqlock_irq(seqlock_t *miSeqlock);
void write_sequnlock_irq(seqlock_t *miSeqlock);
void write_seqlock_bh(seqlock_t *miSeqlock);
void write_sequnlock_bh(seqlock_t *miSeqlock);
unsigned int read_seqbegin(seqlock_t *miSeqlock);
int read_seqretry(seqlock_t *miSeqlock, unsigned int iv);
unsigned int read_seqbegin_irqsave(seqlock_t *miSeqlock, long flags);
int read_seqretry_irqrestore(seqlock_t *miSeqlock, unsigned int iv, long flags);
```

6.2 Read-Copy-Update (Leer-Copiar-Actualizar)

Los Spinlocks funcionan bien en la mayoría de las situaciones, pero tienen un costo asociado. Tomar un lock lleva tiempo, especialmente en máquinas SMP, donde la línea de la cache de cada procesador que contiene al lock debe actualizarse o invalidarse cada vez que su valor se modifique. Es por esto que los desarrolladores interesados en la escalabilidad del kernel han dedicado mucho tiempo a buscar alternativas, como la que se describe aqui.

Read-copy-update (RCU) es una técnica de exclusión mutua que puede operar sin bloqueos la mayor parte del tiempo. Puede significar un incremento considerable de la performance cuando:

* los datos a ser protegidos son accedidos a través de un puntero
* la lectura es frecuente
* los cambios son poco usuales
* y las referencias al dato no son mantenidas mientras el KCP duerme

La idea básica detras de RCU es que cuando un dato necesita ser modificado, un puntero a una nueva estructura conteniendo el dato actualizado puede reemplazar al anterior inmediatamente. La estructura con el dato antiguo puede ser liberada luego de que se garantize que ningún proceso en el sistema mantiene una referencia a ella.

El primer paso para usar RCU es definir una estructura que contenga el dato a ser protegido. Las estructuras necesitan ser alocadas dinámicamente y accedidas por un puntero. RCU no puede ser usado con estructuras estáticas.

El código que lea estructuras de datos protegidas por RCU sólo necesitan tomar algunas precauciones simples:

- Debería llamarse a rcu_read_lock() antes de acceder al dato y a rcu_read_unlock() despues de accederlo. Ésta llamada sólo deshabilita la capacidad de preemption, éste es un paso rápido y necesario para que RCU funcione correctamente.
- El KCP no debe dormir mientras el "lock de lectura RCU" sea mantenido.

Por lo tanto, el código que lee un dato protegido por RCU podría ser similar a éste:

```
struct estructuraDatos *punteroDatos;
rcu_read_lock();
punteroDatos = BuscarDatos(args...);
hacer_algo(punteroDatos);       /* No debe dormir */
rcu_read_unlock();
```

La versión de escritura de RCU es un poco más complicada. Para actualizar una estructura de datos, el código comienza alocando una nueva copia de esa estructura y llenándola con información. Luego debe reemplazarse el puntero a la estructura desactualizada con el nuevo puntero, manteniendo una copia del primero. Despues de ésta operación, cualquier KCP corriendo en otro procesador encontrará la nueva versión de la estructura.

Sin embargo, el puntero antiguo no puede ser liberado todavía, ya que es posible que otro procesador lo siga utilizando. RCU ofrece una forma de avisar en que momento es seguro liberar esa memoria. Ésto se logra mediante la función call_rcu():

```
void call_rcu(struct rcu_head *head,
        void (*func)(void *arg),
        void *arg);
```

Debe proveerse una estructura rcu_head como parámetro, pero no es necesario inicializarla antes de la llamada a call_rcu. Generalmente, ésta estructura está embebida en la estructura de datos a ser protegida por RCU. "func" es lo que suele denominarse una función "callback". Cuando la estructura pueda ser liberada de manera segura, la función func será invocada por el kernel, enviándole a args como argumentos. Normalmente, todo lo que func necesita hacer es llamar a kfree() o alguna de sus variantes (liberan memoria alocada por el kernel).

El algoritmo RCU procede esperando hasta que cada procesador en el sistema haya

sido planificado al menos una vez. Ya que por definición se requiere que las referencias a las estructuras protegidas por RCU no sean mantenidas durante las dormidas, ningún procesador puede mantener una referencia a una estructura antigua despues de haber sido planificado.

Por lo tanto, despues de que el puntero antiguo haya sido reemplazado y todos los procesadores hayan sido planificados al menos una vez, no pueden existir referencias a la estructura anterior, y ésta puede ser liberada.

Como nota adicional, el subsistema RCU exporta la funcionalidad de "esperar a que todos hayan sido planificados", que podría ser útil en algún otro contexto. Para realizar esta espera, sólo es necesario hacer una llamada a la función synchronize_kernel().

6.3 El kernel preemptible

La propiedad de preemptible implica que la ejecución de código kernel puede ser interrumpido cuando el scheduler decida que hay algo más importante que hacer. Sin embargo hay una excepción a tener en cuenta: no se producirá una preemption si el código ejecutándose actualmente mantiene un spinlock. Por lo tanto las precauciones para asegurar la exclusión mutua en un entorno SMP también funciona en este caso.

En algunos casos especiales puede ser necesario controlar directamente la posibilidad de que se produzcan preemptions (por ejemplo, cuando se acceden a las denominadas "variables por CPU"). Para ésto, dos macros incluidas en <linux/preempt.h> pueden ser útiles: preempt_disable() y preempt_enable(). Si se quisiera rehabilitar la posibilidad de preemption pero no se desea ser “expulsado” del procesador inmediatamente, se puede usar preempt_enable_no_resched().

Normalmente, la re-planificación por preemption toma lugar sin ningún esfuerzo por parte del código que está siendo “expulsado”. Sin embargo, cuando una rutina dura un tiempo considerable puede querer verificar explícitamente si existe alguna re-planificación pendiente (por ejemplo, cuando la rutina alcanza un punto donde dormir por un tiempo no sería tan costoso). Entonces puede usarse preempt_check_resched().

Algo interesante que se logró en la última versión del kernel es que ahora es mucho más fácil saber si una parte específica del código del kernel está ejecutándose en una sección crítica. Una única variable en la estructura struct task del proceso current almacena los estados de preemption, interrupciones y funciones diferibles. Una nueva macro, in_atomic(), verifica todos estos estados y retorna un valor distinto de cero si el kernel está ejecutando código que debería completarse sin interrupciones. Entre otras cosas, esta macro se usa para interceptar llamadas desde contextos atómicos a funciones que podrían dormir.

7.Referencia Bibliográfica

- “Linux Device Drivers, Third Edition”, Jonathan Corbet, Alessandro Rubini, and Greg

Kroah-Hartman, O'Reilly & Associates.

- “Understanding the Linux Kernel, Second Edition ”, D. P. Bovet & M. Cesati, O'Reilly & Associates.
- “http://lwn.net/Kernel/”
- “http://www.kerneltrap.org”