

SINCRONIZZAZIONE DEL KERNEL

Si può pensare al kernel come ad un server che risponde a richieste; esse provengono o da processi in esecuzione sulla CPU o da dispositivi esterni che inviano interrupt. L'analogia serve a sottolineare che parti del kernel non sono eseguite sequenzialmente, ma intercalate tra di loro. Perciò possono dare origine a "race conditions"¹ (letteralmente "corse critiche") che vanno controllate con tecniche appropriate di sincronizzazione.

Come il kernel soddisfa le richieste

Il kernel può essere considerato come un cameriere che deve soddisfare due tipi di richieste: quelle che provengono da clienti normali e quelle che provengono da un limitato numero di capi. La politica adottata è la seguente:

1 – Se un capo chiama il cameriere mentre è disoccupato, questi inizia a soddisfare le richieste del capo.

2 – Se un capo chiama il cameriere mentre è occupato con un cliente, questi sospende il lavoro e inizia a soddisfare la richiesta del capo.

3 – Se un capo chiama il cameriere mentre è occupato con un altro capo, questi sospende il lavoro e inizia a soddisfare la richiesta del secondo capo. Quando ha finito, riprende il lavoro per il primo capo.

4 – Un capo può interrompere il cameriere mentre è occupato con un cliente. Dopo aver soddisfatto la richiesta del capo, il cameriere può decidere di lasciar perdere temporaneamente le richieste del primo cliente e prestare attenzione ad un altro.

I servizi del cameriere corrispondono al codice eseguito in modalità del kernel. Se la CPU lavora in modalità utente, il cameriere è considerato disoccupato.

Le richieste dei capi corrispondono agli interrupt, mentre quelle dei clienti corrispondono alle chiamate di sistema o alle eccezioni in modalità utente. I processi in modalità utente che vogliono richiedere servizi al kernel devono eseguire un'istruzione apposita (nei sistemi 80x86 una `int 0x80` o una `sysenter`). Essa origina un'eccezione che costringe la CPU a passare in modalità del kernel; per questo verranno definite "eccezioni" sia le chiamate di sistema che le eccezioni vere e proprie.

Le regole ai punti 1, 2 e 3 si possono paragonare all'esecuzione annidata di kernel control paths (KCP). La regola al punto 4 corrisponde ad una delle novità più interessanti del kernel 2.6, la *preemption*.

Preemption del kernel

E' difficile dare una buona definizione di *preemption*. Si può dire in prima battuta che un kernel è dotato di *preemption* quando può avvenire una commutazione di contesto in cui il processo sostituito sta eseguendo una funzione del kernel, cioè opera in modalità del kernel. Purtroppo in Linux le cose sono più complicate:

- sia in un kernel dotato che non dotato di *preemption* un processo in modalità del kernel può cedere volontariamente la CPU, ad esempio perché sta aspettando una risorsa. Una commutazione

¹ Situazione che si verifica quando due o più processi stanno leggendo o scrivendo qualche dato condiviso, e il risultato finale dipende da come sono schedulati i processi.

di processo di questo tipo si definisce *pianificata*. Comunque un kernel dotato differisce da uno non dotato di preemption nel modo in cui un processo in modalità del kernel reagisce agli eventi asincroni che inducono ad una commutazione di contesto, ad esempio ad un gestore di interrupt che risveglia un processo a priorità più alta. Questa commutazione si definisce *forzata*.

- Le commutazioni di processo vengono eseguite dalla macro `switch_to`. Sia in un kernel dotato che non dotato di preemption, una commutazione avviene quando un processo ha portato a termine l'attività di qualche thread del kernel e lo scheduler è stato chiamato. Però in un kernel non dotato di preemption il processo corrente non può essere sostituito a meno che non stia per passare in modalità utente.

Perciò la caratteristica principale di un kernel dotato di preemption è che un processo in modalità del kernel può essere sostituito da un altro processo anche nel corso dell'esecuzione di una funzione del kernel.

Per esemplificare: mentre A esegue un gestore di eccezioni (necessariamente in modalità del kernel), il processo B, con priorità più alta, diviene eseguibile perché viene ricevuta una IRQ (Interrupt ReQuest) e il gestore corrispondente risveglia B. Se il kernel è dotato di preemption, una commutazione forzata sostituisce A con B. Il gestore di eccezione rimane in sospeso e viene riattivato solo quando lo scheduler lo seleziona ancora. Se il kernel non è dotato di preemption, non avviene alcuna commutazione di processo finché A non termina l'esecuzione del gestore di eccezioni oppure rilascia volontariamente la CPU.

Oppure si consideri un processo che esegue un gestore di eccezioni ed il suo tempo di esecuzione si esaurisce. Se il kernel è dotato di preemption, il processo viene rimpiazzato immediatamente; se no, il processo continua la sua esecuzione fino al termine del gestore di eccezioni oppure fino a quando rilascia volontariamente la CPU.

Il motivo principale per adottare la preemption è ridurre la *latenza di esecuzione* dei processi in modalità utente, cioè il tempo che intercorre tra l'istante in cui divengono eseguibili e l'istante in cui iniziano l'esecuzione. I processi che svolgono compiti schedulati in base al tempo (come controller di dispositivi esterni, monitor ambientali, riproduttori di filmati video) traggono vantaggio dalla preemption, perché essa riduce il rischio che vengano ritardati da altri processi in modalità del kernel.

Introdurre la preemption nel kernel 2.6 non ha richiesto un cambiamento drastico nell'architettura interna rispetto ai kernel precedenti. Essa è disabilitata quando il valore del campo `preempt_count` del descrittore `thread_info` ottenuto dalla macro `current_thread_info()` è maggiore di 0. Il campo racchiude tre diversi contatori, per cui è maggiore di 0 quando si verifica una delle seguenti condizioni:

- 1 – il kernel sta eseguendo una ISR (Interrupt Service Routine);
- 2 – le funzioni differibili sono disabilite (sempre vero quando il kernel sta eseguendo un `softirq` o un `tasklet`);
- 3 – la preemption è stata esplicitamente disabilitata impostando il contatore ad un valore positivo.

Le regole indicano che il kernel è soggetto a preemption solo quando sta eseguendo un gestore di eccezioni (in particolare una chiamata di sistema) e la preemption non è stata esplicitamente disabilitata. In più, la CPU deve avere gli interrupt locali abilitati, altrimenti la preemption non può essere abilitata.

Alcune macro operano sul contatore di preemption del campo `preempt_count`:

Macro	Descrizione
<code>preempt_count()</code>	Seleziona il campo <code>preempt_count</code> nel descrittore <code>thread_info</code>

Macro	Descrizione
preempt_disable()	Incrementa di 1 il valore del contatore
preempt_enable_no_resched()	Decrementa di 1 il valore del contatore
preempt_enable()	Come sopra; in più chiama preempt_schedule() se il flag TIF_NEED_RESCHED è attivato
get_cpu()	Come preempt_disable(), ma restituisce anche il numero della CPU locale
put_cpu()	Come preempt_enable()
put_cpu_no_resched()	Come preempt_enable_no_resched()

La macro preempt_enable() decrementa il contatore, poi verifica se il flag TIF_NEED_RESCHED è attivato. In questo caso esiste una richiesta di commutazione di processo in sospeso, perciò la macro chiama preempt_schedule(), che esegue le azioni seguenti:

```
struct thread_info *ti = current_thread_info();
/*
 * If there is a non-zero preempt_count or interrupts are disabled,
 * we do not want to preempt the current task. Just return..
 */
if (likely(ti->preempt_count || irqs_disabled()))
    return;
do {
    add_preempt_count(PREEMPT_ACTIVE);
    schedule();
    sub_preempt_count(PREEMPT_ACTIVE);
    /*
     * Check again in case we missed a preemption opportunity
     * between schedule and now.
     */
    barrier();
} while (unlikely(test_thread_flag(TIF_NEED_RESCHED)));
```

La funzione controlla se gli interrupt locali sono abilitati e preempt_count vale 0; se entrambe le condizioni sono vere, chiama schedule() per selezionare un altro processo. Perciò la preemption del kernel può avere luogo quando un KCP (di solito un gestore di interrupt) termina o quando un gestore di eccezioni la abilita di nuovo per mezzo di preempt_enable(), oppure quando le funzioni differibili sono abilitate.

Da notare che la preemption introduce un carico addizionale non trascurabile; per questo Linux 2.6 ha un'opzione di configurazione che consente di disabilitarla in fase di compilazione.

Quando la sincronizzazione è necessaria

I concetti di *race condition* e *regioni critiche* si applicano sia ai processi che ai KCP. Una race condition può verificarsi quando il risultato di una elaborazione dipende da come sono nidificati due o più KCP

interconnessi fra loro. Una regione critica è una porzione di codice che deve essere portata a termine dal KCP che la esegue prima che ad essa possa accedere un altro KCP.

I KCP interconnessi complicano la vita dello sviluppatore: deve fare molta attenzione per identificare nei gestori di eccezioni e di interrupt, nelle funzioni differibili e nei thread del kernel le regioni critiche. Esse vanno protette in modo da assicurare che solo un KCP alla volta vi possa accedere.

Se due diversi gestori di interrupt devono accedere alla stessa struttura che contiene alcune variabili correlate tra loro (ad esempio un buffer e un intero che ne indica la lunghezza), tutte le istruzioni che le riguardano devono essere poste in un'unica regione critica. Se il sistema ha un'unica CPU, la regione critica può essere protetta disabilitando gli interrupt prima di permettere l'accesso alle strutture, perché la nidificazione del KCP si ha solo quando gli interrupt sono abilitati.

D'altra parte, se alle stesse strutture accedono solo le routine di servizio delle chiamate di sistema e vi è solo una CPU, la regione critica può essere protetta semplicemente disabilitando la preemption del kernel prima di consentire l'accesso ad essa.

Le cose sono molto più complesse nei sistemi multiprocessore. Le CPU possono eseguire codice del kernel contemporaneamente, per cui lo sviluppatore non può essere certo che una struttura dati sia accessibile in modo sicuro solo perché la preemption è disabilitata, e alla struttura non ha mai accesso un gestore di interrupt, eccezioni o di softirq.

Il kernel offre perciò una vasta scelta di tecniche di sincronizzazione. E' compito dello sviluppatore risolvere ogni singolo problema mediante la tecnica più efficace.

Quando la sincronizzazione non è necessaria

Alcune scelte di progetto semplificano in qualche modo la sincronizzazione dei KCP:

- Tutti i gestori di interrupt confermano l'interrupt al PIC e disabilitano la linea di IRQ. Ulteriori interrupt dello stesso tipo non possono verificarsi prima del termine del gestore.
- Gestori di interrupt, softirq e tasklet sono sia non bloccanti che non soggetti a preemption, per cui non possono venire sospesi per un lungo periodo di tempo. Nel caso peggiore, la loro esecuzione può essere rimandata di poco a causa del verificarsi di altri interrupt (esecuzione nidificata del KCP).
- Un gestore di interrupt non può essere interrotto da un KCP che esegue una funzione differibile o una routine di servizio di una chiamata di sistema.
- Softirq e tasklet non possono intercalarsi su una data CPU.
- Lo stesso tasklet non può essere eseguito contemporaneamente su più CPU.

Ognuna di queste scelte progettuali può essere vista come un vincolo che porta a codificare in modo più semplice alcune funzioni del kernel. Seguono alcuni esempi di possibili semplificazioni:

- gestori di interrupt e tasklet non devono essere rientranti;
- variabili specifiche per CPU a cui accedono solo softirq e tasklet non richiedono sincronizzazione;
- una struttura a cui accede un solo tipo di tasklet non richiede sincronizzazione.

D'ora in poi saranno trattati i casi che richiedono sincronizzazione al fine di prevenire la corruzione dei dati a causa di un accesso non sicuro a strutture condivise.

Primitive di sincronizzazione

Viene ora esaminato come intercalare KCP evitando race conditions. In tabella sono elencate le tecniche usate dal kernel di Linux. La colonna "scope" (campo di visibilità) indica se la tecnica di sincronizzazione si applica ad una sola o a tutte le CPU del sistema. Ad esempio disabilitare gli interrupt locali interessa solo una CPU; al contrario una operazione atomica riguarda tutte le CPU (operazioni atomiche su diverse CPU non possono intercalarsi mentre accedono alle stesse strutture dati).

tecnica	descrizione	scope
Variabili specifiche per CPU	Duplicazione delle strutture per le CPU	Tutte le CPU
Operazioni atomiche	Istruzioni atomiche di lettura-modifica-scrittura per un contatore	Tutte le CPU
Barriere di memoria	Impedire il riordino delle istruzioni	Tutte le CPU o locale
Spin lock	Blocco (lock) con attesa attiva (busy wait)	Tutte le CPU
Semafori	Blocco con attesa bloccante (sospensione)	Tutte le CPU
Seqlock	Blocco basato su un contatore di accesso	Tutte le CPU
Disabilitare interrupt locali	Vietare la gestione degli interrupt su una singola CPU	locale
Disabilitare softirq locali	Vietare la gestione delle funzioni differibili su una singola CPU	locale
Read-copy update (RCU)	Accesso senza blocchi a strutture condivise tramite puntatori	Tutte le CPU

Vengono brevemente descritte le singole tecniche. In seguito verrà mostrato come combinare queste tecniche per proteggere le strutture dati.

Variabili specifiche per CPU

La miglior tecnica di sincronizzazione consiste nel progettare il kernel in modo da evitare la necessità di sincronizzazione; infatti ogni primitiva di sincronizzazione ha un costo in termini di prestazioni.

La tecnica più semplice è dichiarare variabili specifiche per CPU. Fondamentalmente si tratta di array con un elemento per ogni CPU del sistema.

Una CPU non deve avere accesso agli elementi dell'array che corrispondono alle altre CPU; però può leggere e modificare liberamente il proprio elemento senza timore di race conditions, dato che è l'unica CPU abilitata a farlo. Ciò significa che tali variabili si possono usare solo in determinati casi – quando ha un senso logico dividere i dati per CPU.

Gli elementi delle variabili sono allineati in memoria in modo che ogni struttura rientri in una linea di cache hardware. Quindi accessi concorrenti non provocano snooping della cache e cancellazioni di linee, operazioni costose in termini di efficienza del sistema.

Le variabili specifiche per CPU proteggono dagli accessi concorrenti delle altre CPU ma non da quelli delle funzioni asincrone (gestori di interrupt e funzioni differibili). In questi casi occorrono altre tecniche. Inoltre sono soggette a race conditions provocate dalla preemption del kernel, sia in sistemi uniprocessore che multiprocessore. Come regola generale, un KCP dovrebbe accedere ad una variabile specifica per CPU con la preemption disabilitata. Si può verificare considerando cosa potrebbe succedere se un KCP ottenesse l'indirizzo della copia locale della variabile e poi venisse sostituito e spostato su un'altra CPU; l'indirizzo si riferirebbe sempre alla variabile della prima CPU.

La tabella elenca funzioni e macro per la gestione delle variabili specifiche per CPU.

Macro o funzione	Descrizione
DEFINE_PER_CPU(type, name)	Alloca staticamente un array chiamato name di strutture di tipo data
per_cpu(name, cpu)	Seleziona l'elemento relativo alla CPU cpu dell'array name
__get_cpu_var(name)	Seleziona l'elemento relativo alla CPU locale dell'array name
get_cpu_var(name)	Disabilita la preemption, poi seleziona l'elemento relativo alla CPU locale dell'array name
put_cpu_var(name)	Abilita la preemption (name non viene usato)
alloc_percpu(type)	Alloca dinamicamente un array di strutture di tipo data e restituisce il suo nome
free_percpu(pointer)	Rilascia un array allocato dinamicamente con indirizzo pointer
per_cpu_ptr(pointer, cpu)	Restituisce l'indirizzo dell'elemento relativo alla CPU cpu dell'array con indirizzo pointer

Operazioni atomiche

Diverse istruzioni assembly sono del tipo “leggi-modifica-scrivi”, cioè accedono due volte alla locazione di memoria, la prima per leggere il vecchio valore, la seconda per scrivere quello nuovo. Se due KCP in esecuzione su due CPU cercano di “leggere-modificare-scrivere” la stessa locazione di memoria nello stesso istante eseguendo operazioni non atomiche, per prima cosa tentano di leggere la locazione. L'arbitro della memoria (il circuito che serializza gli accessi ai chip della RAM) permette l'accesso ad una di esse e rinvia quello dell'altra. Quando la prima lettura è terminata, la seconda CPU legge anch'essa lo stesso dato (vecchio). Entrambe le CPU poi tentano di scrivere lo stesso (nuovo) valore; ancora una volta l'accesso è serializzato dall'arbitro e le due operazioni di scrittura si susseguono in ordine. Il risultato finale è errato perché le due CPU scrivono lo stesso valore (nuovo); perciò le operazioni intercalate hanno lo stesso effetto di una sola.

Il metodo migliore per prevenire race conditions dovute ad operazioni “leggi-modifica-scrivi” è assicurare che tali operazioni siano “atomiche” a livello di chip. Esse vanno eseguite con una singola istruzione senza interruzioni ed evitando accessi alla stessa locazione da parte di altre CPU. Queste “operazioni atomiche” sono la base di altre tecniche per creare regioni critiche.

Nel set di istruzioni del processore 80x86:

- le istruzioni che non eseguono accessi di memoria o ne eseguono uno solo allineato sono atomiche;

- le istruzioni “leggi-modifica-scrivi” (come inc o dec) sono atomiche se nessun altro processore impegna il bus di memoria dopo la lettura e prima della scrittura. Questo è sempre vero in un sistema uniprocessore;
- le istruzioni “leggi-modifica-scrivi” il cui opcode è preceduto dal byte lock (0xf0) sono atomiche anche in sistemi multiprocessore. Quando l'unità di controllo legge il prefisso, blocca il bus di memoria fino al termine dell'istruzione, impedendo l'accesso ad altri processori;
- le istruzioni il cui opcode è preceduto dal byte rep (0xf2, 0xf3, che obbligano l'unità di controllo a ripetere alcune istruzioni diverse volte) non sono atomiche. L'unità di controllo verifica la presenza di interrupt in sospeso prima di compiere un'altra iterazione.

Scrivendo codice C non si può garantire che il compilatore usi un'istruzione atomica per un'operazione come $a = a + 1$ oppure $a++$. Perciò il kernel Linux offre un tipo `atomic_t` (un contatore accessibile in modo atomico) e varie macro e funzioni che operano su variabili `atomic_t` e sono implementate da istruzioni assembly atomiche (ad es. `atomic_read(v)`, `atomic_set(v, i)`, `atomic_add(i, v)` ecc.). Nei sistemi multiprocessore, queste istruzioni hanno il prefisso `lock`). Un'altro insieme di operazioni atomiche agisce sulle maschere di bit (`test_bit(nr, addr)`, `set_bit(nr, addr)` ecc.).

Barriere di ottimizzazione e barriere di memoria

Quando si usano compilatori ottimizzati, non si può essere sicuri che le istruzioni vengano eseguite nello stesso ordine in cui sono presenti nel codice sorgente. Ad esempio, un compilatore potrebbe riordinare le istruzioni assembly per ottimizzare l'uso dei registri. In più, le CPU moderne eseguono varie istruzioni in parallelo e possono riordinare l'accesso alla memoria. Questi accorgimenti possono aumentare la velocità dei programmi.

Per quanto riguarda la sincronizzazione, invece, il riordino va evitato. Le cose possono farsi complicate se un'istruzione posta dopo una primitiva di sincronizzazione viene eseguita prima di questa. Perciò tutte le primitive di sincronizzazione agiscono come barriere di ottimizzazione e di memoria.

Una *barriera di ottimizzazione* assicura che le istruzioni assembly corrispondenti alle istruzioni C inserite prima di essa non vengano mescolate dal compilatore a quelle poste dopo la barriera. In Linux, la macro `barrier()` si espande in

```
__asm__ __volatile__("" : : "memory")
```

Essa agisce come una barriera di ottimizzazione. L'istruzione `asm` indica al compilatore di inserire un frammento di codice assembly (in questo caso vuoto). La parola chiave `volatile` impedisce al compilatore di mescolare l'istruzione `asm` con altre istruzioni del programma. La parola chiave `memory` obbliga il compilatore a considerare che tutte le locazioni RAM sono state cambiate dal codice assembly; di conseguenza il compilatore non può ottimizzare il codice usando i valori di locazioni di memoria contenuti nei registri della CPU prima dell'istruzione `asm`. Da notare che la barriera di ottimizzazione non assicura che l'esecuzione delle istruzioni assembly non vengano mescolate dalla CPU – questo è il compito di una barriera di memoria.

Una *barriera di memoria* assicura che le operazioni precedenti la barriera vengano terminate prima di iniziare quelle seguenti. La barriera si comporta come un firewall che non può essere oltrepassato dalle istruzioni assembly. Nei processori 80x86 le seguenti istruzioni assembly agiscono come barriere di memoria perché “serializzano” le istruzioni:

- tutte quelle che operano sulle porte di I/O;
- tutte quelle con il byte di prefisso lock;
- tutte quelle che scrivono nei registri di controllo, di sistema, di debug (ad esempio cli e sti che agiscono sul flag IF del registro eflags);
- le istruzioni lfence, sfence, mfence, introdotte nel Pentium 4 per implementare le barriere di memoria in lettura, in scrittura e in lettura-scrittura rispettivamente;
- alcune istruzioni specializzate, come iret, che conclude un interrupt o un gestore di eccezioni.

Linux usa pochi tipi di barriera di memoria, elencate in tabella. Esse agiscono anche come barriere di ottimizzazione, dato che il compilatore non deve spostare le istruzioni rispetto alla barriera. Le barriere in lettura agiscono solo sulle istruzioni che leggono la memoria, mentre le barriere in scrittura agiscono solo su quelle che scrivono in memoria. Le barriere sono utili nei sistemi uniprocessore e multiprocessore. Le primitive smp_xxx() sono impiegate solo su sistemi multiprocessore; in quelli a processore singolo non fanno nulla. Le altre possono essere impiegate in entrambi i tipi di sistema.

Macro	Descrizione
mb()	Barriera di memoria per MP e UP
rmb()	Barriera di memoria in lettura per MP e UP
wmb()	Barriera di memoria in scrittura per MP e UP
smp_mb()	Barriera di memoria solo per MP
smp_rmb()	Barriera di memoria in lettura solo per MP
smp_wmb()	Barriera di memoria in scrittura solo per MP

L'implementazione delle barriere dipende dall'architettura del sistema. Sui processori 80x86 rmb() è definita come

```
asm volatile("lfence" : : "memory")
```

se il sistema supporta l'istruzione assembly lfence; altrimenti è definita come

```
asm volatile("lock; addl $0,0(%%esp)" : : "memory")
```

L'istruzione asm inserisce un frammento di codice assembly nel codice generato dal compilatore e agisce come barriera di ottimizzazione; l'istruzione tra parentesi aggiunge 0 alla locazione di memoria in cima allo stack; di per sé è inutile, ma il prefisso lock fa sì che diventi una barriera di memoria per la CPU.

La macro wmb() è ancora più semplice perché si espande in barrier(). Questo perché i processori Intel esistenti non riordinano mai gli accessi di memoria in scrittura, per cui non è necessario serializzare le istruzioni. La macro comunque impedisce al compilatore di rimescolare le istruzioni.

Da notare che in tutti i sistemi multiprocessore, tutte le operazioni atomiche descritte in precedenza agiscono come barriere di memoria perché hanno il prefisso lock.

Spin lock

Una tecnica di sincronizzazione molto usata è il “lock” (blocco). Quando un KCP deve accedere ad una struttura condivisa o entrare in una regione critica, deve acquisire il lock. Una risorsa protetta da un meccanismo di lock è come se fosse dentro ad una stanza con la porta chiusa a chiave. Per poter aprire la porta il KCP deve trovare la serratura sbloccata. Questo accade solo se la risorsa è libera. Poi, per tutto il tempo in cui usa la risorsa, la porta rimane chiusa a chiave. Quando il KCP rilascia la risorsa, la porta viene sbloccata e un altro KCP può entrare.

Gli spin lock sono un tipo particolare di lock, progettati per operare in sistemi multiprocessore. Se il KCP trova lo spin lock aperto, acquisisce il lock e continua l'esecuzione. Se invece lo trova bloccato da un KCP in esecuzione su un'altra CPU, gli “gira intorno” (spin) eseguendo ripetutamente un ciclo di attesa, finché il blocco non viene rilasciato.

Il ciclo dello spin lock rappresenta una forma di “attesa attiva” (busy wait). Il KCP in attesa continua l'esecuzione anche se non ha nulla da fare, se non far trascorrere il tempo. Nonostante ciò, gli spin lock sono di solito convenienti, perché molte risorse sono bloccate solo per frazioni di millisecondo; sarebbe un dispendio di tempo molto maggiore rilasciare la CPU per riprendere l'esecuzione in seguito.

Come regola generale, la preemption del kernel è disabilitata in ogni regione critica protetta da spin lock. In sistemi uniprocessore i lock di per sé sono inutili, e le primitive di spin lock hanno solo la funzione di disabilitare o abilitare la preemption. Da notare che la preemption è abilitata durante la fase di attesa attiva, per cui un processo in attesa di uno spin lock può essere sostituito da un processo a più alta priorità.

In Linux uno spin lock è rappresentato da una struct `spinlock_t` con due campi:

lock: codifica lo stato dello spin lock: il valore 1 corrisponde a “sbloccato”, mentre 0 e i valori negativi indicano “bloccato”;

break_lock: flag che indica se un processo è in attesa attiva per il lock (presente solo se il kernel supporta sia SMP che preemption).

Sei macro sono impiegate per inizializzare, verificare ed impostare gli spin lock; si basano su operazioni atomiche: questo assicura che gli spin lock vengano aggiornati correttamente anche in caso di accessi multipli su diverse CPU.

Macro	Descrizione
<code>spin_lock_init()</code>	Imposta lo spin lock a 1 (sbloccato)
<code>spin_lock()</code>	Esegue un ciclo finché lo spin lock diventa 1, poi lo imposta a 0 (bloccato)
<code>spin_unlock()</code>	Imposta lo spin lock a 1 (sbloccato)
<code>spin_unlock_wait()</code>	Attende fino a che lo spin lock diventa 1 (sbloccato)
<code>spin_is_locked()</code>	Restituisce 0 se lo spin lock è impostato a 1, altrimenti restituisce 1
<code>spin_trylock()</code>	Imposta a 0 lo spin lock; restituisce 1 se il valore precedente era 1, altrimenti 0

La macro `spin_lock` con preemption del kernel

La macro `spin_lock()` è usata per acquisire uno spin lock. La descrizione che segue si riferisce a un kernel con `preemption` in un sistema SMP. La macro riceve l'indirizzo `slp` dello spin lock come parametro ed esegue le azioni seguenti:

1 – Chiama `preempt_disable()` per disabilitare la `preemption` del kernel.

2 – Chiama la funzione `_raw_spin_trylock()` che esegue una operazione atomica di “test-and-set” sul campo `slock`; questa funzione esegue dapprima le istruzioni equivalenti a questo frammento di codice:

```
movb $0, %al
xchgb %al, slp->slock
```

L'istruzione assembly `xchg` scambia in modo atomico il contenuto del registro a 8 bit `%al` (che contiene 0) con la locazione di memoria puntata da `slp->slock`. Restituisce 1 se il valore di `slock` (che ora si trova in `%al`) è positivo, altrimenti restituisce 0.

3 – Se il vecchio valore dello spin lock è positivo, termina: il KCP ha acquisito lo spin lock.

4 – In caso contrario, il KCP non può acquisire lo spin lock, quindi la macro deve attendere che venga rilasciato. Chiama `preempt_enable()` per annullare l'incremento del contatore di `preemption` eseguito al punto 1. Se la `preemption` era abilitata prima dell'esecuzione della macro, un altro processo può rimpiazzare il processo in attesa dello spin lock.

5 – Se il campo `break_lock` vale 0, lo imposta a 1. Controllando questo valore, il processo che possiede il lock ed è in esecuzione su un'altra CPU, può sapere se ci sono altri processi in attesa del lock. Se un processo lo detiene per lungo tempo, può decidere di rilasciarlo in anticipo per consentire ad un processo in attesa di continuare l'esecuzione.

6 – Esegue il ciclo di attesa

```
while (spin_is_locked(slp) && slp->break_lock)
    cpu_relax();
```

La macro `cpu_relax()` si riduce ad una istruzione assembly `pause`; essa è stata introdotta nel Pentium 4 per ottimizzare il ciclo degli spin lock. Introducendo un piccolo ritardo, velocizza l'esecuzione del ciclo che segue il lock e riduce il consumo di potenza. L'istruzione `pause` è compatibile con i modelli più vecchi perché corrisponde a `rep;nop`.

7 – Torna al punto 1 per tentare ancora di acquisire il lock.

La macro `spin_lock()` senza `preemption` del kernel

Se la `preemption` non è stata abilitata alla compilazione, la macro è differente: contiene codice assembly simile al frammento seguente²:

```
1:    lock; decb slp->slock
      jns 3f
2:    pause
```

² L'implementazione attuale è più complessa. Il codice all'etichetta 2, eseguito solo se il lock è bloccato, è incluso in una sezione ausiliaria, in modo che, nel caso più frequente di lock sbloccato, la cache hardware non viene riempita da codice che non viene quasi mai eseguito. Qui i dettagli di ottimizzazione sono stati trascurati.

```
    cmpb $0, slp->slock
    jle 2b
    imp 1b
```

3:

L'istruzione `dec b` decrementa il valore dello spin lock; l'istruzione è atomica perché ha il prefisso `lock`. Viene poi eseguito un test sul flag: se è azzerato, significa che lo spin lock era impostato a 1 (sbloccato), così che la normale esecuzione continua con l'etichetta 3 (il suffisso `f` indica che l'etichetta è "forward", cioè compare più avanti nel programma). Altrimenti il ciclo all'etichetta 2 (il suffisso `b` indica che si tratta di una etichetta "backward", cioè precedente) viene eseguito fino a che lo spin lock ritorna positivo. Poi l'esecuzione riparte dall'etichetta 1, dato che non è sicuro procedere senza controllare se un altro processore ha acquisito il lock.

La macro `spin_unlock`

La macro `spin_unlock` rilascia uno spin lock acquisito in precedenza; esegue il codice seguente:

```
    movb $1, slp->slock
```

Poi chiama `preempt_enable()` (che, se la preemption non è abilitata, non fa nulla). Da notare che non viene usato il byte lock perché gli accessi alla memoria in sola scrittura sono sempre eseguiti atomicamente dai processori 80x86.

Spin lock read/write

Gli spin lock in lettura/scrittura sono stati introdotti per aumentare il livello di parallelismo all'interno del kernel. Permettono a diversi KCP di leggere contemporaneamente le stesse strutture dati, finché nessuno di essi le modifica. Se un processo vuole scrivere entro una struttura, deve acquisire la versione in scrittura del lock, che garantisce un accesso esclusivo alla risorsa. Di sicuro permettere una lettura contemporanea incrementa le performance del sistema.

Ogni spin lock in lettura/scrittura è rappresentato da una struttura `rwlock_t`; `lock` è un campo di 32 bit che codifica due distinte informazioni:

- un contatore di 24 bit contiene il numero di KCP che stanno leggendo le strutture protette. Il complemento a due del valore del contatore è contenuto nei bit 0-23 del campo;
- un flag `unlock` che viene attivato quando nessun KCP sta leggendo o scrivendo, e azzerato nel caso contrario. Il flag è contenuto nel bit 24 del campo.

Il campo `lock` contiene il valore `0x01000000` se lo spin lock è libero (flag `unlock` attivato e nessun lettore), il valore `0x00000000` se è stato acquisito in scrittura (flag `unlock` azzerato e nessun lettore) e ogni altro numero nella sequenza `0x00ffffff`, `0x00fffffe`, ecc. se è stato acquisito in lettura da uno, due o più processi (flag `unlock` azzerato e il complemento a 2 del numero di processi in lettura). Come `spinlock_t`, anche `rwlock_t` contiene un campo `break_lock`.

La macro `rwlock_init` inizializza il campo `lock` a `0x01000000` (sbloccato) e il campo `break_lock` a 0.

Acquisire e rilasciare un lock in lettura

La macro `read_lock`, applicata all'indirizzo `rwlp` di uno spin lock in lettura/scrittura, è simile alla macro `spin_lock` descritta in precedenza. Se il kernel è stato compilato con `preemption`, la macro compie le stesse azioni di `spin_lock()`, con una differenza: per acquisire il lock in lettura/scrittura al punto 2 esegue la funzione `_raw_read_trylock()`:

```
static inline int _raw_read_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock;
    atomic_dec(count);
    if (atomic_read(count) >= 0)
        return 1;
    atomic_inc(count);
    return 0;
}
```

Si accede al campo `lock` per mezzo di una operazione atomica. Da notare, però, che l'intera funzione non agisce atomicamente sul contatore; ad esempio, il contatore può cambiare dopo aver controllato il suo valore con l'istruzione `if` e prima di restituire 1. Nonostante ciò, la funzione svolge il suo lavoro; infatti restituisce 1 solo se il contatore era maggiore di zero prima del decremento, poiché risulta uguale a `0x01000000` in caso non ci siano processi, `0x00ffffff` se c'è un processo in lettura e `0x00000000` se ce n'è uno in scrittura.

Se il kernel è stato compilato senza `preemption`, la macro `read_lock` contiene il seguente codice assembly:

```
    movl $rwlp->lock, %eax
    lock; subl $1, (%eax)
    jns 1f
    call __read_lock_failed
1:
```

Dove `__read_lock_failed()` contiene le istruzioni seguenti:

```
__read_lock_failed:
    lock; incl (%eax)
1: pause
    cmpl $1, (%eax)
    js 1b
    lock; decl (%eax)
    js __read_lock_failed
    ret
```

La macro `read_lock` decrementa atomicamente di 1 il valore dello spin lock, incrementando così il numero dei lettori. Lo spin lock viene acquisito se il decremento porta a un valore non negativo; altrimenti viene invocata la `__read_lock_failed()`. La funzione incrementa atomicamente il campo `lock` per annullare il decre-

mento effettuato in precedenza e poi compie un ciclo fino a quando il valore diventa positivo (maggiore o uguale a 1). Poi tenta di acquisire lo spin lock (un altro KCP potrebbe acquisirlo subito prima dell'istruzione `cmp`). Rilasciare lo spin lock è semplice, dato che la macro `read_unlock` deve solo incrementare il contatore nel campo `lock` con l'istruzione:

```
lock; incl rwp->lock
```

per ridurre il numero di lettori, e poi invocare `preempt_enable()`.

Acquisire e rilasciare un lock in lettura

La macro `write_lock` è implementata come le precedenti. Se il kernel è configurato con `preemption`, la funzione la disabilita e prova ad acquisire il lock chiamando `_raw_write_trylock()`. Se questa funzione restituisce 0, il lock è già stato acquisito e la macro riabilita la `preemption` e inizia un ciclo di attesa.

```
static inline int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

La funzione sottrae `0x01000000` dal valore dello spin lock, azzerando così il flag `unlock` (bit 24). Se il risultato è zero (nessun lettore), il lock è acquisito e la funzione restituisce 1; altrimenti la funzione aggiunge atomicamente `0x01000000` al valore dello spin lock per annullare l'operazione precedente.

Ancora una volta, rilasciare lo spin lock è molto più semplice perché la macro `write_unlock` deve solo attivare il flag `unlock`

```
lock; addl $0x01000000, rwp
```

e poi chiamare `preempt_enable()`.

Seqlock

Quando si usano gli spin lock in lettura/scrittura, le richieste dei KCP in lettura o scrittura hanno la stessa priorità: il processo in lettura deve attendere che quello in scrittura abbia terminato e viceversa.

I *seqlock* introdotti in Linux 2.6 sono simili agli spin lock in lettura/scrittura, ma offrono una maggiore priorità ai processi in scrittura: essi possono procedere anche se sono attivi dei processi in lettura. L'aspetto positivo è che un processo in scrittura non attende mai (a meno che non ce ne sia un altro in scrittura); l'aspetto negativo è che talvolta un processo in lettura deve rileggere varie volte gli stessi dati per avere una copia valida.

Ogni *seqlock* è una struttura `seqlock_t` formata da due campi: un campo `lock` di tipo `spinlock_t` e un intero `sequence`. Questo secondo campo ha il ruolo di un contatore di sequenza. Ogni processo in lettura deve leggere il contatore due volte, prima e dopo la lettura dei dati, e controllare che i valori coincidano. In caso

siano diversi, un processo in scrittura ha iniziato la sua attività e ha incrementato il contatore di sequenza, informando così il processo in lettura che i dati appena letti non sono validi.

Una variabile `seqlock_t` è inizializzata a “sbloccata” assegnandole il valore `SEQLOCK_UNLOCKED` oppure eseguendo la macro `seqlock_init`. I processi in scrittura acquisiscono e rilasciano il lock chiamando `write_seqlock()` e `write_sequnlock()`. La prima funzione acquisisce lo spin lock nella struttura `seqlock_t`, poi incrementa di uno il contatore; la seconda incrementa ancora il contatore, poi rilascia lo spin lock. Questo garantisce che quando il processo è in fase di scrittura, il contatore sia dispari, e quando nessuno sta alterando i dati in scrittura, il contatore sia pari. Una regione critica in lettura:

```
unsigned int seq;
do {
    seq = read_seqbegin(&seqlock);
    /* regione critica */
} while (read_seqretry(&seqlock, seq));
```

`read_seqbegin()` restituisce il numero corrente di sequenza del `seqlock`; `read_seqretry()` restituisce 1 sia se il valore della variabile locale `seq` è dispari (un processo in scrittura stava aggiornando le strutture quando la funzione `read_seqbegin()` è stata invocata), sia se il valore di `seq` non coincide con il valore corrente del contatore di sequenza (un processo in scrittura ha iniziato l'attività mentre uno in lettura stava eseguendo il codice nella regione critica). Da notare che quando un processo in lettura entra in una regione critica, non ha necessità di disabilitare la preemption; d'altra parte, un processo in scrittura disabilita automaticamente la preemption entrando nella regione critica, poiché acquisisce lo spin lock.

Non tutti i tipi di struttura possono essere protetti da un `seqlock`. Come regola generale, bisogna considerare queste condizioni:

- le strutture non devono includere puntatori che vengano modificati da processi in scrittura e dereferenziati da processi in lettura (altrimenti si potrebbero modificare puntatori sotto il naso dei processi in lettura);

- il codice nella regione critica dei processi in lettura non deve avere effetti collaterali (altrimenti letture multiple avrebbero effetti diversi rispetto ad una sola).

Inoltre, le regioni critiche dei processi in lettura dovrebbero essere brevi e i processi in scrittura dovrebbero acquisire raramente i `seqlock`, altrimenti ripetuti accessi in lettura potrebbero provocare elevati sovraccarichi. Un esempio di impiego di `seqlock` in Linux 2.6 è nella protezione di alcune strutture correlate al sistema di gestione del tempo.

Ready-Copy Update (RCU)

E' un'altra tecnica di sincronizzazione per proteggere strutture dati a cui accedono in lettura varie CPU. RCU permette a diversi processi in lettura e scrittura di lavorare contemporaneamente (si tratta di un miglioramento rispetto ai `seqlock`, che consentono l'accesso multiplo solo in lettura). In più RCU è senza blocchi, dato che non usa né lock né contatori condivisi dalle CPU; è un grande vantaggio su spin lock e `seqlock` che hanno un sovraccarico dovuto allo snooping e all'invalidazione delle cache. L'idea di base sta nel limitare il campo di visibilità di un RCU:

- 1 - possono essere protette da RCU solo le strutture allocate dinamicamente e referenziate da puntatori;

- 2 - nessun KCP può rimanere in attesa in una regione critica protetta da RCU.

Quando un KCP vuole leggere una struttura protetta da RCU, esegue la macro `rcu_read_lock()`, equivalente a `preempt_disable()`; poi dereferenzia il puntatore alla struttura e inizia a leggere. Il processo non può essere sospeso fino a che non ha finito la lettura; la fine della regione critica è identificata dalla macro `rcu_read_unlock()`, equivalente alla `preempt_enable()`.

Poiché il processo in lettura fa molto poco per prevenire race conditions, ci si può aspettare che il processo in scrittura debba fare di più. Infatti esso deve dereferenziale il puntatore e fare una copia dell'intera struttura, poi può modificare la copia. Una volta terminata l'operazione, il processo cambia il puntatore alla struttura in modo che punti alla copia modificata. Dato che modificare il puntatore è un'operazione atomica, ogni processo in lettura o scrittura vede entrambe le copie, sia la nuova che la vecchia; non si può perciò avere corruzione dei dati. Comunque serve una barriera di memoria per garantire che il puntatore modificato sia visibile alle altre CPU solo dopo che la struttura è stata modificata. Tale barriera è implicitamente introdotta se alla RCU viene affiancato uno spin lock per evitare l'accesso concorrente in scrittura.

Il vero problema con RCU è che la vecchia copia della struttura non può essere rilasciata prima che venga aggiornato il puntatore. Infatti i processi che hanno iniziato la lettura mentre la struttura viene modificata in scrittura potrebbero essere ancora in fase di lettura della vecchia copia. Essa può essere rilasciata solo dopo che tutti i processi (potenziali) in lettura hanno eseguito la macro `rcu_read_unlock()`. Il kernel richiede a tutti i potenziali processi in lettura di eseguire la macro prima che:

- la CPU compia una commutazione di processo;
- la CPU passi alla modalità utente;
- la CPU esegua il ciclo idle.

In ognuno di questi casi diciamo che la CPU ha attraversato uno *stato quiescente*.

La funzione `call_rcu()` viene chiamata dal processo in scrittura per liberarsi della vecchia copia della struttura. Riceve come parametri l'indirizzo di un descrittore `rcu_head` (di solito incorporato nella struttura da rilasciare) e l'indirizzo di una funzione di *call-back*³ da chiamare quando tutte le CPU hanno attraversato lo stato quiescente. Una volta eseguita, la funzione di call-back solitamente rilascia la copia della struttura.

La `call_rcu()` memorizza nel descrittore `rcu_head` l'indirizzo della funzione di call-back e i suoi parametri, poi inserisce il descrittore in una lista di call-back per-CPU. Periodicamente, una volta ogni tick, il kernel controlla se la CPU locale ha attraversato uno stato quiescente. Quando tutte le CPU locali lo hanno fatto, un tasklet locale – il cui descrittore è memorizzato nella variabile per-CPU `rcu_tasklet` – esegue tutte le call-back della lista.

RCU è una novità di Linux 2.6; viene usato nel networking e nel filesystem virtuale.

Semafori

I semafori sono una primitiva di sincronizzazione che permette ai processi in attesa di rimanere quiescenti finché la risorsa diviene disponibile. Attualmente Linux offre due tipi di semafori:

- semafori del kernel, usati dai KCP;
- semafori del tipo System V IPC, usati dai processi in modalità utente.

³ Il termine si può rendere con funzione di richiamo; si riferisce ad una routine di servizio chiamata da una funzione; di solito viene passata come argomento alla funzione chiamante.

In questa sezione vengono trattati i semafori del primo tipo, d'ora in avanti definiti semplicemente semafori.

Un semaforo è simile a uno spin lock, in quanto non permette al KCP di procedere finché il lock non è sbloccato. Comunque, quando un KCP tenta di acquisire una risorsa protetta da un semaforo, il processo corrispondente viene sospeso e riprende l'esecuzione quando la risorsa viene sbloccata. Perciò i semafori (del kernel) possono essere acquisiti solo da funzioni che possono essere sospese; gestori di interrupt e funzioni differibili non possono usarli. Un semaforo è un oggetto di tipo struct *semaphore* che contiene i campi seguenti:

count: contiene un valore atomic_t. Se è maggiore di 0, la risorsa è disponibile. Se è uguale a 0, il semaforo è occupato ma non ci sono altri processi in attesa. Se è negativo, il semaforo è occupato e almeno un altro processo è in attesa.

wait: contiene l'indirizzo di una lista di attesa di tutti i processi quiescenti in attesa della risorsa. Se count è maggiore o uguale a 0, la lista è vuota.

sleepers: contiene un flag che indica se qualche processo è quiescente in attesa del semaforo.

Le funzioni `init_MUTEX()` e `init_MUTEX_LOCKED()` possono essere usate per inizializzare un semaforo per un accesso esclusivo: impostano `count = 1` (risorsa libera con accesso esclusivo) e `count = 0` (risorsa occupata con accesso esclusivo garantito al processo che inizializza il semaforo), rispettivamente. Le macro `DECLARE_MUTEX` e `DECLARE_MUTEX_LOCKED` eseguono la stessa operazione, ma in più allocano staticamente la variabile struct *semaphore*. Da notare che un semaforo può essere inizializzato con un valore positivo arbitrario `n` per `count`. In questo caso almeno `n` processi possono accedere alla risorsa.

Acquisire e rilasciare un semaforo

Rilasciare un semaforo è molto più semplice che acquisirlo. Per farlo, un processo chiama la funzione `up()`, sostanzialmente equivalente a questo codice:

```
movl $sem->count, %ecx
lock; incl(%ecx)
jg 1f
lea %ecx, %eax
pushl %edx
pushl %ecx
call __up
popl %ecx
popl %edx
```

1:

mentre `__up()` è la seguente:

```
__attribute__((regparm(3))) void __up(struct semaphore *sem)
{
    wake_up(&sem->wait);
}
```

La funzione `up()` incrementa il campo `count` del semaforo `*sem`, poi controlla se il suo valore è maggiore di zero. Queste operazioni devono essere eseguite atomicamente per evitare che un altro KCP possa accedere

allo stesso valore con conseguenze disastrose. Se `count > 0`, non ci sono processi in attesa e non occorre fare altro. Se `count ≤ 0`, viene chiamata `__up()` per risvegliare un processo in attesa. Da notare che `__up` riceve i suoi parametri attraverso il registro `eax`.

Quando un processo vuole acquisire il semaforo, chiama la funzione `down()`. La sua implementazione è equivalente a:

down:

```
movl $sem->count, %ecx
lock; decl(%ecx)
jns 1f
lea %ecx, %eax
pushl %edx
pushl %ecx
call __down
popl %ecx
popl %edx
```

1:

mentre `__down()` è la seguente:

```
__attribute__((regparm(3))) void __down(struct semaphore *sem) {
    DECLARE_WAITQUEUE(wait, current);
    unsigned long flags;
    current->state = TASK_UNINTERRUPTIBLE;
    spin_lock_irqsave(&sem->wait.lock, flags);
    add_wait_queue_exclusive_locked(&sem->wait, &wait);
    sem->sleepers++;
    for(;;) {
        if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
            sem->sleepers = 0;
            break;
        }
        sem->sleepers = 1;
        spin_unlock_irqrestore(&sem->wait.lock, flags);
        schedule();
        spin_lock_irqsave(&sem->wait.lock, flags);
        current->state = TASK_UNINTERRUPTIBLE;
    }
    remove_wait_queue_locked(&sem->wait, &wait);
    wake_up_locked(&sem->wait);
    spin_unlock_irqrestore(&sem->wait.lock, flags);
    current->state = TASK_RUNNING;
}
```

La funzione `down()` decrementa il campo `count` di `*sem` e controlla se il valore è negativo; queste operazioni sono eseguite in modo atomico. Se `count ≥ 0`, il processo corrente acquisisce la risorsa e la sua esecuzione continua normalmente. Se `count < 0`, il processo corrente va sospeso. Il contenuto di alcuni registri viene salvato sullo stack e viene chiamata `__down()`. Questa funzione cambia lo stato di `current` da `TASK_RUNNING` a `TASK_UNINTERRUPTIBLE` e lo inserisce nella coda di attesa del semaforo. Prima di accedere alle strutture di `*sem`, acquisisce il lock `sem->wait.lock`, che protegge la coda di attesa del semaforo, e disabilita

gli interrupt locali. Di solito le funzioni che operano sulle code di attesa acquisiscono e rilasciano lo spin lock quando inseriscono e cancellano un elemento. La funzione `__down()` usa lo spin lock anche per proteggere gli altri campi della struttura semaphore da processi concorrenti su altre CPU. Per questo scopo usa la versione "locked" delle funzioni che operano sulle code di attesa, che presuppongono che lo spin lock sia già stato acquisito prima della loro chiamata.

`__down()` sospende `current` fino a che il semaforo non è rilasciato, ma lo fa in modo contorto. Per capire il codice, occorre tener presente che il campo `sleepers` vale 0 se non ci sono processi in coda, altrimenti vale 1. Si considerino alcuni casi tipici:

Semaforo MUTEX aperto ($count = 1$, $sleepers = 0$). La macro `down()` imposta `count` a 0 e salta alla istruzione successiva del programma principale: `__down()` non viene eseguita.

Semaforo MUTEX chiuso senza processi in attesa ($count = 0$, $sleepers = 0$). La macro `down()` decrementa `count` e chiama la funzione `__down()` con `count = -1` e `sleepers = 0`. Ad ogni iterazione del ciclo, la funzione controlla se `count` è negativo (da notare che `count` non viene modificato da `atomic_add_negative()` perché `sleepers = 0`).

- Se `count` è negativo, chiama `schedule()` per sospendere `current`. `count` è ancora -1 , mentre `sleepers = 1`. Il processo continua poi il ciclo e ripete il controllo.
- Se `count` non è negativo, la funzione imposta `sleepers` a 0 ed esce dal ciclo. Tenta di risvegliare un altro processo in coda (in questo caso però la coda è vuota) e termina acquisendo il semaforo. Uscendo, sia `count` che `sleepers` sono impostati a 0, come richiesto dal caso in cui il semaforo è chiuso ma non ci sono processi in coda.

Semaforo MUTEX chiuso con altri processi in attesa ($count = -1$, $sleepers = 1$). La macro `down()` decrementa `count` e chiama la funzione `__down()` con `count = -2` e `sleepers = 1`. La funzione imposta temporaneamente `sleepers = 2`, poi annulla il decremento effettuato dalla macro aggiungendo il valore `sleepers-1` a `count`. Allo stesso tempo controlla se `count` è ancora negativo (il semaforo potrebbe essere stato rilasciato appena prima che `__down()` entrasse nella regione critica).

- Se `count` è negativo, riporta `sleepers` a 1 e chiama `schedule` per sospendere `current`. Ora `count = -1` e `sleepers = 1`.
- Se `count` non è negativo, imposta `sleepers` a 0, tenta di risvegliare un altro processo in coda ed esce con il semaforo acquisito. Uscendo, `count` viene impostato a 0 e `sleepers` a 0. Il valore dei campi appare errato, poiché ci sono altri processi in coda. Tuttavia si consideri che un altro processo è già stato risvegliato; esso fa una iterazione del ciclo; `atomic_add_negative()` sottrae 1 da `count` riportandolo a -1 ; in più, il processo risvegliato, prima di tornare a riposo, resetta `sleepers` a 1.

Perciò il codice lavora bene in ogni caso. Va considerato che `wake_up()` risveglia un solo processo alla volta, perché i processi in coda sono esclusivi.

Solo i gestori di eccezioni e particolari routine di servizio di chiamate di sistema usano la funzione `down()`. I gestori di interrupt e le funzioni differibili non lo possono fare perché sospende il processo quando il semaforo è occupato. Per questo scopo Linux usa `down_trylock()` per tutte le funzioni asincrone ricordate prima. E' identica a `down()`, ma ritorna immediatamente se la risorsa è impegnata.

Esiste anche una funzione leggermente diversa chiamata `down_interruptible()`. Viene usata soprattutto dai driver di dispositivo perché consente ai processi che ricevono un segnale mentre sono bloccati al semaforo di effettuare l'operazione "down". Se il processo quiescente viene svegliato da un segnale prima di acquisire la risorsa, la funzione incrementa il valore di `count` del semaforo e restituisce il valore `-EINTR`. D'altra par-

te se `down_interruptible()` giunge normalmente a termine e ottiene la risorsa, restituisce 0. Il driver deve abortire l'operazione di I/O quando riceve il valore `-EINVAL`.

Infine, dato che i processi generalmente trovano i semafori aperti, le funzioni sono ottimizzate per questo caso. In particolare la funzioni `up()` e `down()` non eseguono istruzioni di salto se la coda di attesa del semaforo è vuota, oppure se il semaforo è aperto, rispettivamente. La maggior parte della complicazione aggiuntiva delle funzioni è dovuta allo sforzo di evitare istruzioni onerose per il flusso di esecuzioni principale.

Semafori in lettura/scrittura

Sono simili agli spin lock in lettura/scrittura descritti in precedenza; la differenza è che il processo viene sospeso invece di attendere in un ciclo che il semaforo sia aperto. Vari KCP possono acquisire il semaforo in lettura contemporaneamente, mentre solo uno può ottenere l'accesso esclusivo in scrittura. Perciò il semaforo può essere acquisito in scrittura solo se nessun altro KCP lo ha ottenuto né in lettura né in scrittura. Questo tipo di semaforo aumenta il parallelismo del kernel e l'efficienza complessiva del sistema.

Il kernel gestisce i processi in attesa di un semaforo in lettura/scrittura in un ordine FIFO (first in first out, il primo ad entrare è il primo ad uscire – cioè una coda; ndt). Ogni processo sia in lettura che in scrittura che trova il semaforo occupato, viene inserito all'ultimo posto della coda di attesa. Quando torna libero, viene controllato il primo processo della lista, che viene sempre risvegliato. Se vuole accedere in scrittura, gli altri processi restano quiescenti; se vuole accedere in lettura, vengono risvegliati anche tutti i restanti processi in attesa per lettura, fino al primo in attesa per scrittura. Ovviamente, i processi in attesa per lettura che seguono quest'ultimo non vengono risvegliati.

Ogni semaforo in lettura/scrittura è descritto da una struttura `rw_semaphore` con i seguenti campi:

`count`: contiene due contatori a 16 bit. Quello nella word più significativa codifica il complemento a due della somma dei processi in scrittura non in attesa (possono essere 0 oppure 1) e dei processi in attesa. Il contatore nella word meno significativa codifica il numero totale di processi non in attesa, sia in lettura che in scrittura.

`wait_list`: punta alla lista dei processi in attesa. Ogni elemento della lista è una struttura `rwsem_waiter` che include un puntatore al descrittore del processo in attesa e un flag che indica se il processo attende in lettura o scrittura.

`wait_lock`: spin lock usato per proteggere la coda di attesa e la struttura `rw_semaphore` in sé.

La funzione `init_rwsem()` inizializza una struttura `rw_semaphore` impostando `count` a 0, `wait_lock` a sbloccato e `wait_list` a lista vuota.

Le funzioni `down_read()` e `down_write()` acquisiscono il semaforo in lettura e scrittura, rispettivamente. Similmente, `up_read()` e `up_write()` lo rilasciano. Le funzioni `down_read_trylock()` e `down_write_trylock()`, a differenza di quelle menzionate in precedenza, non bloccano il processo se il semaforo è occupato. Infine la `downgrade_write()` trasforma in modo atomico un lock in scrittura in uno in lettura. La loro implementazione segue lo schema di quelle dei semafori normali.

Completions

Linux utilizza un'altra primitiva di sincronizzazione simile ai semafori detta *completions* (completamenti). Sono stati introdotti per risolvere una race condition che può verificarsi nei sistemi multiprocessore quando

un processo A alloca un semaforo temporaneo, lo inizializza come MUTEX chiuso, passa il suo indirizzo a B e poi chiama `down()` su di esso. A ha intenzione di distruggere il semaforo appena viene risvegliato. Più tardi B, in esecuzione su un'altra CPU, chiama `up()` sul semaforo. Nell'implementazione attuale, `up()` e `down()` possono agire in modo concorrente sullo stesso semaforo. Perciò A può essere risvegliato e distruggere il semaforo mentre B sta ancora eseguendo `up()` su di esso. Come conseguenza, `up()` potrebbe tentare di accedere a una struttura che non esiste più. Sicuramente sarebbe possibile cambiare l'implementazione di `up()` e `down()` per impedire un accesso concorrente; questo però significherebbe maggiore complessità per funzioni impiegate frequentemente.

Completion è un tecnica di sincronizzazione creata per risolvere questo problema. La struttura completion comprende una coda di attesa e un flag:

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

La funzione che corrisponde a `up()` è chiamata `complete()`. Riceve come argomento l'indirizzo di una struttura completion, chiama `spin_lock_irqsave()` sullo spin lock della coda di completion, incrementa il campo `done`, risveglia il processo esclusivo presente nella coda `wait` e finalmente chiama `spin_unlock_irqrestore()`.

La funzione che corrisponde a `down()` è chiamata `wait_for_completion()`. Riceve come argomento l'indirizzo di una struttura completion e controlla il valore di `done`: se è maggiore di 0, termina, perché `complete()` è già stata eseguita su un'altra CPU. In caso contrario, aggiunge `current` alla fine della coda di attesa come processo esclusivo e lo sospende nello stato `TASK_UNINTERRUPTIBLE`. Una volta risvegliato, la funzione rimuove `current` dalla coda; poi controlla il valore di `done`: se è uguale a 0, la funzione termina, altrimenti `current` viene sospeso di nuovo. Anche `wait_for_completion()` fa uso dello spin lock della coda di attesa di completion.

La reale differenza fra completion e semaforo sta in come viene usato lo spin lock della coda di attesa. In completion, lo spin lock viene usato per garantire che `complete()` e `wait_for_completion()` non vengano eseguite in modo concorrente. Nei semafori, lo spin lock è usato per evitare che `down()`, eseguita contemporaneamente, metta disordine nella struttura semaphore.

Disabilitare gli interrupt locali

E' la tecnica chiave per garantire che una parte di codice del kernel sia trattata come una regione critica. Permette ad un KCP di continuare l'esecuzione anche quando un dispositivo hardware invia un IRQ, fornendo così una reale protezione delle strutture a cui hanno accesso anche i gestori di interrupt. Di per sé non protegge dall'accesso concorrente alle strutture da parte dei gestori di interrupt di altre CPU, per cui nei sistemi multiprocessore questa tecnica viene affiancata da spin lock.

La macro `local_irq_disable()`, che fa uso dell'istruzione assembly `cli`, disabilita gli interrupt sulla CPU locale. La macro `local_irq_enable()`, che fa uso dell'istruzione assembly `sti`, li abilita. Queste due istruzioni rispettivamente azzerano e attivano il flag IF del registro `eflag`. La macro `irqs_disabled()` restituisce 1 se `IF = 0`, 0 se `IF = 1`.

Quando il kernel entra in una regione critica, disabilita gli interrupt azzerando il flag IF. Alla fine della regione, spesso non può semplicemente attivarlo di nuovo. Gli interrupt possono essere nidificati, per cui il kernel non conosce il valore di IF precedente l'esecuzione del KCP corrente. In questi casi il KCP deve salvare lo stato del flag e ripristinarlo alla fine. Queste operazioni sono svolte per mezzo di `local_irq_save()` e

local_irq_restore()). La prima macro copia il valore del registro eflag in una variabile locale; il flag IF viene poi azzerato con un'istruzione cli. Al termine della regione critica, la seconda macro ripristina il valore di eflag: perciò gli interrupt sono abilitati solo se lo erano prima della chiamata di local_irq_save()).

Disabilitare ed abilitare le funzioni differibili

Le funzioni differibili vengono eseguite in momenti non definiti, di solito al termine dei gestori di interrupt. Perciò le loro strutture vanno protette da race conditions. Un modo banale per impedire l'esecuzione di tali funzioni è disabilitare gli interrupt su quella CPU; dato che non vengono attivati i gestori di interrupt, non vengono avviate le funzioni asincrone dei softirq.

A volte però il kernel deve disabilitare le funzioni differibili lasciando abilitati gli interrupt. Lo fa agendo sul contatore di softirq contenuto nel campo preempt_count del descrittore thread_info. Va ricordato che do_softirq() non esegue mai softirq se il contatore è positivo. Inoltre i tasklet sono implementati su softirq, per cui assegnando al contatore un valore positivo si disabilita su una CPU l'esecuzione di tutte le funzioni differibili.

La macro local_bh_disable() aggiunge 1 al contatore softirq della CPU locale, mentre local_bh_enable() sottrae 1. Il kernel può usare diverse chiamate nidificate della local_bh_disable(); le funzioni differibili verranno riabilitate solo se le chiamate di local_bh_enable() risulteranno di pari numero.

Dopo aver decrementato il contatore, local_bh_enable() compie due importanti operazioni per garantire un'esecuzione tempestiva di processi in attesa da lungo tempo:

1 – controlla i contatori hardirq e softirq del campo preempt_count della CPU locale; se entrambi valgono 0 e ci sono softirq in attesa chiama do_softirq() per attivarli;

2 – controlla se il flag TIF_NEED_RESCHED è attivato; se c'è una richiesta di commutazione di processo in attesa, chiama preempt_schedule().

Sincronizzazione degli accessi alle strutture del kernel

Una struttura condivisa può essere protetta da race condition usando una delle tecniche di sincronizzazione descritte. Le prestazioni del sistema dipendono dal tipo di tecnica adottata. Di solito gli sviluppatori del kernel adottano questa regola: *mantenere il più alto livello di concorrenza possibile nel sistema*. Questo livello dipende da due fattori principali:

- dal numero di dispositivi di I/O che operano contemporaneamente;
- dal numero di CPU usate nell'elaborazione.

Per massimizzare il flusso di dati, gli interrupt andrebbero disabilitati per periodi brevi; infatti mentre gli interrupt sono disabilitati, le IRQ dei dispositivi sono temporaneamente ignorate dal PIC e nessuna attività può avere luogo nei dispositivi.

Per usare le CPU in modo efficiente le tecniche basate su spin lock devono essere evitate il più possibile. Quando una CPU compie cicli di attesa del lock, spreca prezioso tempo macchina. Inoltre gli spin lock hanno effetti ancora più negativi sullo stato delle cache hardware.

Seguono due esempi di come si può evitare la sincronizzazione pur mantenendo alto il livello di concorrenza:

- una struttura composta da un solo valore intero può essere manipolata dichiarandola di tipo `atomic_t` e usando operazioni atomiche, che sono più veloci rispetto a spin lock e a disabilitazione degli interrupt, e inoltre rallentano solo i KCP che vogliono accedere a quella struttura;
- inserire un elemento in una lista collegata condivisa non è mai un'operazione atomica, perché consiste in almeno due assegnamenti di puntatori. Nonostante ciò il kernel talvolta può compiere questa operazione senza disabilitare gli interrupt o usare lock. Si consideri ad esempio il caso di una routine di servizio di una chiamata di sistema che deve inserire un elemento in una lista a collegamento singolo mentre un gestore di interrupt sta scandendo la lista in modo asincrono.

Nel linguaggio C l'inserimento si realizza con queste istruzioni:

```
new->next = list_element->next
list_element->next = new
```

In assembly il tutto si riduce a due istruzioni atomiche consecutive. La prima imposta il puntatore `next` dell'elemento `new`, ma non modifica la lista. Perciò se il gestore di interrupt scandisce la lista tra la prima e la seconda istruzione, non vede l'elemento `new`. Se lo fa dopo la seconda, vede anche il nuovo elemento. L'importante è che in entrambi i casi la lista è in uno stato consistente e non corrotto. Comunque l'integrità è assicurata solo se il gestore di interrupt non modifica la lista. Se lo fa, il puntatore `next` nell'elemento `new` non è più valido.

Comunque gli sviluppatori devono garantire che l'ordine delle due assegnazioni non venga alterato dal compilatore o dalla CPU, altrimenti se il gestore interrompesse la routine tra la prima e la seconda operazione, troverebbe una lista corrotta. E' necessaria in questo caso una barriera di memoria in scrittura:

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

Scelta tra spin lock, semafori e disabilitare gli interrupt

Sfortunatamente, i modelli di accesso alle strutture del kernel sono molto più complessi dei casi precedenti e gli sviluppatori sono costretti ad usare le primitive di sincronizzazione. Di solito la scelta dipende dal tipo di KCP che vuole avere accesso ai dati, come mostrato in tabella. Da ricordare che quando un KCP acquisisce uno spin lock, disabilita gli interrupt locali o i softirq, la preemption è automaticamente disabilitata.

Tipo di KCP	Sistemi uniprocessore	Sistemi multiprocessore
Eccezioni	Semafori	Nessuno
Interrupt	Disabilitare interrupt	Spin lock
Funzioni differibili	Nessuno	Nessuno o spin lock
Eccezioni + interrupt	Disabilitare interrupt locali	Spin lock
Eccezioni + funzioni differibili	Disabilitare softirq locali	Spin lock
Interrupt + funzioni differibili	Disabilitare interrupt locali	Spin lock

Tipo di KCP	Sistemi uniprocessore	Sistemi multiprocessore
Eccezioni + interrupt + funzioni differibili	Disabilitare interrupt locali	Spin lock

Protezione di strutture a cui hanno accesso eccezioni

Quando l'accesso alle strutture è limitato alle eccezioni, le race conditions sono semplici da capire e da prevenire. Le eccezioni più comuni che danno problemi di sincronizzazione sono le routine di servizio delle chiamate di sistema, nelle quali la CPU opera in modalità del kernel per soddisfare una richiesta di un processo in modalità utente. In questo caso una struttura di solito rappresenta una risorsa da assegnare a due o più processi.

Le race conditions sono evitate usando semafori, perché sospendono i processi fino a che la risorsa diviene disponibile. Da notare che i semafori lavorano bene sia nei sistemi uniprocessore che in quelli multiprocessore. Neppure la preemption del kernel dà problemi. Se un processo che ha acquisito un semaforo viene pre-rilasciato, un nuovo processo sulla stessa CPU può tentare di acquisire il semaforo. Quando ciò accade, viene sospeso e il vecchio processo può eventualmente essere eseguito e rilasciare il semaforo. L'unico caso in cui la preemption deve essere disabilitata esplicitamente è l'accesso alle variabili specifiche per CPU.

Protezione di strutture a cui hanno accesso interrupt

Se ad una struttura deve avere accesso solo la "prima parte" di un gestore di interrupt, sapendo che ogni gestore è serializzato rispetto a sé stesso, cioè non può essere eseguita più di una istanza alla volta, non sono richieste tecniche di sincronizzazione.

Le cose sono diverse se alla struttura possono accedere diversi gestori di interrupt. Un gestore può interromperne un altro e in sistemi multiprocessore vari gestori possono essere eseguiti contemporaneamente. Senza sincronizzazione, le strutture verrebbero corrotte rapidamente.

In sistemi uniprocessore, vengono disabilitati gli interrupt in tutte le regioni critiche del gestore. Ciò è sufficiente, dato che nessun'altra primitiva può svolgere questo compito: un semaforo blocca il processo, per cui non è adatto ad un gestore di interrupt; uno spin lock può bloccare il sistema: se il gestore che ha accesso alla struttura viene interrotto, non è più in grado di rilasciare il semaforo, mentre il nuovo gestore rimane in attesa attiva nel ciclo dello spin lock.

Nei sistemi multiprocessore, ci sono più variabili da considerare. Le race condition non possono essere evitate semplicemente disabilitando gli interrupt. Infatti, anche se sono disabilitati su una CPU, possono essere eseguiti su un'altra. Il metodo migliore è disabilitare gli interrupt locali (così che altri gestori non interferiscano sulla CPU locale) e insieme acquisire uno spin lock (semplice o in lettura/scrittura) per proteggere le strutture. In questo caso il sistema non rischia di bloccarsi perché, anche se un gestore eseguito su un'altra CPU trova lo spin lock bloccato, il gestore che lo detiene lo sblocca sulla prima CPU.

Linux usa numerose macro che abbinano il controllo degli interrupt con uno spin lock; nei sistemi uniprocessore queste macro agiscono solo su interrupt e preemption del kernel

Protezione di strutture a cui hanno accesso funzioni differibili

La protezione in questo caso dipende dal tipo di funzione. In precedenza si è visto che softirq e tasklet sono differenti per grado di concorrenza. Prima di tutto, non esistono race conditions nei sistemi uniprocessore, perché la loro esecuzione è sempre serializzata su una CPU, per cui una funzione differibile non può essere interrotta da un'altra. Nei sistemi multiprocessore le race conditions sono possibili:

Tipo di funzione	protezione
softirq	Spin lock
Un tasklet	Nessuna
Diversi tasklet	Spin lock

Una struttura accessibile ai softirq va sempre protetta, di solito con spinlock, perché lo stesso softirq può essere eseguito su varie CPU. Nel caso di un singolo tasklet, non sono necessarie protezioni perché due tasklet dello stesso tipo non possono essere eseguiti contemporaneamente. Diverso il caso in cui vari tasklet possono accedere alla stessa struttura.

Protezione di strutture a cui hanno accesso eccezioni ed interrupt

In sistemi uniprocessore la prevenzione di race condition è molto semplice, perché i gestori di interrupt non sono rientranti e non possono essere interrotti da eccezioni. E' sufficiente disabilitare gli interrupt e, se alla struttura può avere accesso un solo tipo di gestore, non è necessario neppure questo.

In sistemi multiprocessore, la disabilitazione degli interrupt va accompagnata da uno spin lock, che costringe all'attesa i KCP in esecuzione su altre CPU. Talvolta è preferibile sostituire un semaforo allo spin lock. Dato che i gestori di interrupt non possono essere sospesi, devono acquisire il semaforo con la funzione `down_trylock()` e il semaforo funziona come uno spin lock. Le routine di servizio delle chiamate di sistema possono sospendere il processo che le ha chiamate quando il semaforo è occupato: per la maggior parte di esse è il comportamento atteso. In questi casi un semaforo è preferibile allo spin lock, perché comporta un livello di concorrenza superiore.

Protezione di strutture a cui hanno accesso eccezioni e funzioni differibili

Questo caso va trattato come il precedente. Infatti le funzioni differibili sono attivate in caso di interrupt e nessuna eccezione può intervenire durante l'esecuzione di una di esse. E' perciò sufficiente abbinare la disabilitazione degli interrupt con uno spin lock.

Attualmente è più che sufficiente: il gestore di eccezioni può disabilitare solo le funzioni differibili invece degli interrupt locali usando `local_bh_disable()`. Infatti l'esecuzione delle funzioni differibili è serializzata. Come al solito, nei sistemi multiprocessore è necessario usare anche uno spin lock.

Protezione di strutture a cui hanno accesso interrupt e funzioni differibili

Il caso è simile a quello di interrupt e gestori di eccezioni. Un interrupt può interrompere una funzione differibile, ma non viceversa, per cui è sufficiente disabilitare gli interrupt quando è in esecuzione una funzione differita. Un gestore di interrupt può accedere liberamente ad una struttura condivisa con una funzione differibile, a patto che essa non sia disponibile anche per altri gestori.

Nei sistemi multiprocessore, bisogna sempre aggiungere uno spin lock.

Protezione di strutture a cui hanno accesso eccezioni, interrupt e funzioni differibili

Come nel caso precedente, è sufficiente disabilitare gli interrupt e acquisire uno spin lock. Non è necessario disabilitare esplicitamente le funzioni differibili, perché esse vengono attivate al termine dei gestori di interrupt; basta disabilitare questi ultimi.

Esempi di prevenzione di race condition

Gli sviluppatori del kernel devono identificare e risolvere i problemi di sincronizzazione; prevenire le race conditions è però un compito difficile perché richiede una chiara visione di come interagiscono le varie componenti del kernel. Vengono quindi esposti alcuni casi tipici.

Contatori di riferimenti

Sono largamente usati nel kernel per evitare race conditions dovute ad allocazione e rilascio di risorse. Un *contatore di riferimenti* è un semplice contatore atomico `atomic_t` associato ad una risorsa specifica, come una pagina di memoria, un modulo o un file. Il contatore è incrementato in modo atomico quando un KCP inizia ad usare la risorsa, e viene decrementato quando termina. Se il contatore vale 0, la risorsa non è più impiegata e, se necessario, può essere rilasciata.

Big Kernel Lock

Nelle prime versioni di Linux veniva spesso usato il *big kernel lock* (chiamato anche global kernel lock – BKL). In Linux 2.0 era praticamente un semplice spin lock in grado di garantire che un solo processore alla volta potesse eseguire programmi in modalità del kernel. I kernel 2.2 e 2.4 divennero molto più flessibili e non si basarono più su un singolo spin lock; piuttosto, un gran numero di strutture del kernel vennero protette con differenti spin lock. Nel kernel 2.6 il BKL viene usato per proteggere parti di codice vecchio, di solito relative al VFS e a vari tipi di filesystem. A partire dalla versione 2.6.11 il BKL è implementato da un semaforo chiamato `kernel_sem` (mentre prima era uno spin lock), che però è più sofisticato di un semplice semaforo.

Ogni descrittore di processo (PD) include un campo `lock_depth` che consente allo stesso processo di acquisire più volte il BKL. Di conseguenza, due richieste consecutive non sospendono il processo, come per un lock normale. Se il processo non ha acquisito il lock, il campo vale -1; in caso contrario, il valore del campo + 1 indica quante volte il lock è stato acquisito. `lock_depth` è fondamentale per consentire a gestori di interrupt ed eccezioni e a funzioni differibili di acquisire il BKL; senza di esso, ogni funzione asincrona che tenta di acquisire il BKL può dare origine a un deadlock⁴ se il processo corrente lo ha già ottenuto.

Le funzioni `lock_kernel()` e `unlock_kernel()` sono usate per ottenere e rilasciare il BKL. La prima equivale a:

```
depth = current->lock_depth +1;
```

⁴ Situazione di stallo che si verifica quando due processi si bloccano a vicenda aspettando che uno esegua una certa azione necessaria all'altro (come rilasciare una risorsa) e viceversa – ndt.

```
if(depth ==0)
    down(&kernel_sem);
current->lock_depth = depth;
```

La seconda equivale a:

```
if(--current->lock_depth < 0)
    up(&kernel_sem);
```

Da notare che le istruzioni if non devono essere eseguite in modo atomico perché lock_depth non è una variabile globale e ogni CPU indirizza un campo del descrittore del proprio processo corrente. Neppure gli interrupt entro la condizione if determinano una race condition. Anche se il nuovo KCP chiama lock_kernel(), deve rilasciare il BKL prima di terminare.

Sorprendentemente, un processo che ha ottenuto il BKL può chiamare schedule() e abbandonare la CPU. La funzione schedule(), comunque, controlla il campo depth del processo da sostituire e, se il valore è 0 o positivo, rilascia automaticamente kernel_sem. Perciò nessun processo che chiama schedule() può conservare il BKL attraverso una commutazione di contesto. Comunque la funzione schedule() acquisisce di nuovo il BKL per il processo sospeso quando viene di nuovo schedulato.

Le cose sono diverse nel caso in cui un processo che ha ottenuto il BKL viene pre-rilasciato ad opera di un altro processo. Fino al kernel 2.6.10 questo non poteva accadere, perché acquisire uno spin lock comportava la disabilitazione automatica della preemption. L'implementazione attuale invece si basa su un semaforo e questo automatismo non è previsto. In realtà, il poter consentire la preemption entro una regione critica protetta dal BKL è stato il motivo principale del cambio di implementazione. Questo infatti ha effetti positivi sui tempi di risposta del sistema.

Quando un processo che detiene il BKL viene pre-rilasciato, schedule() non deve rilasciare il semaforo, perché il processo sospeso non ha chiesto volontariamente una commutazione di processo, per cui se il BKL venisse rilasciato e un altro processo lo ottenesse, si avrebbe la possibile corruzione delle strutture a cui ha avuto accesso il processo pre-rilasciato.

Per evitare che questo perda il BKL, la funzione preempt_schedule_irq() imposta temporaneamente il campo lock_depth a -1. Controllando questo valore, schedule() ne deduce che il processo non ha acquisito il BKL e perciò non rilascia il kernel_sem, che quindi continua ad essere in possesso del processo sospeso. Quando quest'ultimo viene selezionato di nuovo per l'esecuzione, preempt_schedule_irq() ripristina il valore originario di lock-depth e lascia che il processo riprenda l'esecuzione nella regione critica protetta dal BKL.

Semafori in lettura/scrittura dei descrittori di memoria

Ogni descrittore di memoria di tipo mm_struct include il proprio semaforo nel campo mmap_sem. Il semaforo protegge il descrittore dalle race condition che possono verificarsi quando un descrittore viene condiviso da vari processi "lightweight".

Se il kernel deve creare o ampliare un'area di memoria per qualche processo, chiama do_mmap() che alloca una nuova struttura vm_area_struct. Il processo può però essere sospeso se non c'è memoria disponibile, e può essere avviato un altro processo che condivide lo stesso descrittore di memoria. Senza un semaforo, ogni operazione del secondo processo che richiede accesso al descrittore (un Page Fault dovuto a Copy on Write) può portare a corruzione della memoria.

Il semaforo è implementato in lettura/scrittura dato che alcune funzioni, come il gestore di Page Fault hanno solo bisogno di scandire i descrittori di memoria.

Semaforo della slab cache list

La lista dei descrittori della cache slab è protetta dal semaforo `cache_chain_sem`, che garantisce un diritto esclusivo di accedere e modificare la lista.

Una race condition è possibile quando `kmem_cache_create()` aggiunge un nuovo elemento alla lista, mentre `kmem_cache_shrink()` e `kmem_cache_reap()` scandiscono sequenzialmente la lista. Comunque queste funzioni non sono mai chiamate durante un interrupt e non possono bloccarsi mentre accedono alla lista. Il semaforo serve sia nei sistemi multiprocessore che in quelli uniprocessore con preemption abilitata.

Semaforo dell'inode

Linux memorizza le informazioni sui file su disco in oggetti di memoria chiamati inode. La struttura corrispondente include il proprio semaforo nel campo `i_sem`.

Si può verificare un gran numero di race condition durante la gestione del filesystem. Infatti un file su disco è una risorsa condivisa dagli utenti, perché tutti processi potenzialmente possono accedervi, cambiarne il nome o la collocazione, distruggerlo o duplicarlo. Un processo che elenca i file di una directory, può essere bloccato, e anche in un sistema uniprocessore, altri processi possono accedere alla stessa directory e modificarne il contenuto. Tutte queste race conditions possono essere prevenute proteggendo i file con il semaforo di inode.

Quando un programma usa due o più semafori, esiste il rischio di deadlock, perché due processi possono terminare aspettando che l'altro rilasci un semaforo. Linux in genere non ha problemi, perché ogni KCP acquisisce di solito un semaforo alla volta. Talvolta comunque può avere la necessità di acquisirne due o più. I semafori degli inode sono soggetti a questo problema: succede ad esempio per la routine di servizio della chiamata di sistema `rename()`. In questa operazione sono coinvolti due inode e due semafori diversi. Per evitare un deadlock le richieste per il semaforo vengono evase nell'ordine predefinito degli indirizzi.