Taller de Sistemas Operativos

Sincronización en el Kernel

Agenda

- Motivación
- Consideraciones generales
- Variables por CPU
- Operaciones atómicas
- Optimizaciones y barreras de memoria
- Spin locks
- Seqlocks
- Read-Copy update
- Semáforos

Motivación

- En aplicaciones de memoria compartida se debe tener especial cuidado con estructuras y recursos que son accedidos en forma concurrente.
- Esto no es una excepción para el núcleo de un sistema operativo.
- Si bien en un sistema monoprocesador se debe tener precauciones en cuanto a la sincronización, esto se ve aumentado en gran medida en sistemas multiprocesadores.
- Ejecutar varios hilos (thread) del núcleo del sistema a la vez en distintos procesadores permite un mejor desempeño del sistema, pero trae el costo de introduccir la sincronización a nivel del sistema.

Motivación

- Soporte para multiprocesamiento simétrico se introdujo en el kernel 2.0.
- A partir del kernel 2.6 el sistema es preemptive.
- A nivel del núcleo de Linux se tiene concurrencia debido a:
  - *Interrupciones.*
  - *Softirq y tasklets.*
  - *Expropiación a nivel del núcleo.*
  - *Bloqueos en el núcleo pueden llevar a bloquear un proceso de usuario.*
  - *Multiprocesamiento simétrico.*

Conceptos generales

- Los sistemas operativos con núcleo preemptivo (expropiativo) son los que permiten que un *thread* de ejecución del núcleo sea sustituido por otro.
- Se definen dos niveles de switch de procesos:
  - *Planned process switch*
    - Un thread decide liberar el procesador debido a que tiene que esperar por algún recurso.
  - *Forced process switch*
    - Un thread que le expropia el procesador a otro debido a una mayor prioridad.
- La característica de un núcleo preemptivo es que cuando está ejecutando un proceso en modo protegido, puede ser reemplazado en la mitad de la ejecución por otro proceso.

Conceptos generales

- Recién en el núcleo 2.6 se realizó un núcleo totalmente preemptivo.
- Los núcleos preemptivos favorecen a los procesos de usuario.
- Linux provee de la primitiva `preempt_disable` o `preempt_enable` que permite estipular zonas de expropiación.
- La expropiación está deshabilitada si el campo `preempt_count` de la estructura `thread_info` es mayor que 0.
- Existen primitivas (`preempt_disable`, `preempt_enable`, `preempt_count`) para manejar la preemption del núcleo.
- El núcleo puede ser interrumpido (preempted) solamente cuando se encuentra ejecutando una excepción (system call) y preemption no ha sido específicamente deshabilitada.

Conceptos generales

- Núcleo de Linux kernel: Es como un servicio que atiende requerimientos.
- Partes del núcleo ejecutan de manera mezclada.
- *kernel control path*: es una secuencia de instrucciones que ejecuta el núcleo en nombre del proceso actual (current).
- Interrupciones o excepciones
  - *Menos contexto que un proceso.*

Herramientas de Sincronización

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Variables por CPU

- El método de variables por CPU (Per-CPU Variables) consiste en declarar variables en base cada procesador.
- Una variable por CPU es un arreglo de alguna estructura de datos en donde cada entrada corresponde a un procesador.
- Cada procesador accede a su campo correspondiente en forma libre.
- Se gana en la paralelización del código a expensas de tener estructuras de datos que ocupan espacio en memoria principal.

Variables por CPU

- No proveen, por si solas, protección contra accesos de funciones asincrónicas (handlers de interrupciones y funciones diferidas).
- También existen problemas cuando el núcleo es preemptivo, ya que un proceso puede ser migrado de procesador.
- Como regla general, un *kernel control path* deberá acceder a variables por CPU con la expropiación deshabilitada.

Variables por CPU

Macro or function name	Description
`DEFINE_PER_CPU(type, name)`	Statically allocates a per-CPU array called `name` of `type` data structures
`per_cpu(name, cpu)`	Selects the element for CPU `cpu` of the per-CPU array `name`
`__get_cpu_var(name)`	Selects the local CPU's element of the per-CPU array `name`
`get_cpu_var(name)`	Disables kernel preemption, then selects the local CPU's element of the per-CPU array `name`
`put_cpu_var(name)`	Enables kernel preemption (`name` is not used)
`alloc_percpu(type)`	Dynamically allocates a per-CPU array of `type` data structures and returns its address
`free_percpu(pointer)`	Releases a dynamically allocated per-CPU array at address `pointer`
`per_cpu_ptr(pointer, cpu)`	Returns the address of the element for CPU `cpu` of the per-CPU array at address `pointer`

Operaciones Atómicas

- Para construir un núcleo simétrico es necesario que ciertas operaciones sean hechas en forma atómicas.
- El *hardware* actual provee de rutinas a nivel de chip que logran este tipo de operaciones.
- Intel provee distintas instrucciones:
  - *Operaciones atómicas que modifican a 0 ó 1 una acceso lugar de memoria.*
  - *Operaciones de lectura-modificación-escritura que son atómicas si nadie utiliza el bus de memoria entre la lectura y escritura.*
  - *Operaciones de lectura-modificación-escritura que son precedidas por la instrucción lock byte (0xf0) son atómicas. Cuando la unidad de control detecta el prefijo, realiza el lock del bus de memoria hasta que la instrucción finalizó.*

Operaciones Atómicas

- Linux provee un tipo especial `atomic_t`, con funciones y macros que están implementadas con instrucciones atómicas.
- En los sistemas multiprocesadores estás instrucciones son precedidas por la instrucción `lock byte`.

Operaciones Atómicas

Function	Description
`atomic_read(v)`	Return `*v`
`atomic_set(v,i)`	Set `*v` to `i`
`atomic_add(i,v)`	Add `i` to `*v`
`atomic_sub(i,v)`	Subtract `i` from `*v`
`atomic_sub_and_test(i, v)`	Subtract `i` from `*v` and return 1 if the result is zero; 0 otherwise
`atomic_inc(v)`	Add 1 to `*v`
`atomic_dec(v)`	Subtract 1 from `*v`
`atomic_dec_and_test(v)`	Subtract 1 from `*v` and return 1 if the result is zero; 0 otherwise
`atomic_inc_and_test(v)`	Add 1 to `*v` and return 1 if the result is zero; 0 otherwise
`atomic_add_negative(i, v)`	Add `i` to `*v` and return 1 if the result is negative; 0 otherwise
`atomic_inc_return(v)`	Add 1 to `*v` and return the new value of `*v`
`atomic_dec_return(v)`	Subtract 1 from `*v` and return the new value of `*v`
`atomic_add_return(i, v)`	Add `i` to `*v` and return the new value of `*v`
`atomic_sub_return(i, v)`	Subtract `i` from `*v` and return the new value of `*v`

Optimizaciones y barreras de memoria

- Los compiladores que optimizan código no garantizan que las operaciones sean ejecutadas en el orden en que fueron escritas en el código fuente.
- Aún más, los procesadores de hoy día ejecutan instrucciones en paralelo y reordenan los accesos a memoria.
- Cuando se trabaja con sincronización es necesario garantizar que el código que se encuentra en la zona crítica se mantenga dentro de ella.
- Una forma es proveer barrera de optimización para las instrucciones se ejecutan dentro de las barreras especificadas.
- En Linux se provee de la función `barrier()` que brinda una barrera de optimización.

Optimizaciones y barreras de memoria

- La barreras de memorias garantizan que las operaciones previas a ella son finalizadas antes de las operaciones que están luego.
- En Intel las operaciones que son una barrera de memoria son:
  - *Instrucciones que operan sobre puertos de E/S.*
  - *Todas las instrucciones que tienen prefijo la instrucción* `lock byte`.
  - *Todas las operaciones que escriben en los registros de control, de sistema o debug.*
  - `lfence`, `sfence` *y* `mfence` *(disponibles en PIV) que implementan las barreras de memoria de lectura, escritura y lectura-escritura respectivamente.*

Optimizaciones y barreras de memoria

- Linux provee un conjunto de primitivas que actúan como barreras de memoria:

Macro	Description
`mb( )`	Memory barrier for MP and UP
`rmb( )`	Read memory barrier for MP and UP
`wmb( )`	Write memory barrier for MP and UP
`smp_mb( )`	Memory barrier for MP only
`smp_rmb( )`	Read memory barrier for MP only
`smp_wmb( )`	Write memory barrier for MP only

Spin Locks

- Una técnica de sincronización bastante utilizada es la de utilizar locks.
- Los locks permiten el acceso controlado a secciones críticas donde se manipulan estructuras de datos compartidas.
- Los Spin Locks están diseñados para ambientes de multiprocesador.
- Al ejecutar un spin lock:
  - *Si el kernel control path lo encuentra libre, adquiere el lock y continúa ejecutando.*
  - *Si está ocupado, queda a la espera de su liberación a través de un "busy waiting".*

Spin Locks

- Los *spin locks* son buenos en circunstancias donde el núcleo accede a un *lock* por milisegundos y la tarea de bloquearse y designar un nuevo proceso para ejecutar es mucho más costosa.
- Útiles para el uso en los manejadores de interrupciones.
- Como regla general, la expropiación es deshabilitada (*preemtion disabled*) dentro de las regiones protegidas por los *spin locks*, pero no cuando se realiza el spin.
- No se podría bloquear mientras se mantiene un *spin lock* ya que los que esperan consumirían demasiado procesador.
- En sistemas uniprocesadores los *spin locks* no tienen sentido ya que están consumiendo el recurso.

Spin Locks

- Las primitivas para trabajar con ellos son:

Macro	Description
`spin_lock_init( )`	Set the spin lock to 1 (unlocked)
`spin_lock( )`	Cycle until spin lock becomes 1 (unlocked), then set it to 0 (locked)
`spin_unlock( )`	Set the spin lock to 1 (unlocked)
`spin_unlock_wait( )`	Wait until the spin lock becomes 1 (unlocked)
`spin_is_locked( )`	Return 0 if the spin lock is set to 1 (unlocked); 1 otherwise
spin_trylock( )	Set the spin lock to 0 (locked), and return 1 if the previous value of the lock was 1; 0 otherwise

- El tipo de los *spin locks* es `spinlock_t`

Spin Locks de Lectura/Escritura

- Linux provee también *read/write spin locks* para lograr mayor concurrencia en el sistema.
- En este caso se permiten la concurrencia en la sección crítica de varios lectores a la vez.
- Los escritores acceden en forma única a la sección crítica.
- La estructura utilizada es `rwlock_t` y el campo `lock` corresponde a un entero de 32 bits.
- Los 24 (0..23) primeros bits corresponden a cuantos lectores están accediendo al *lock.*
- El bit 24 está en 0 si el *lock* está ocupado y en 1 si está libre.

Seqlocks

- Fueron introducidos en el núcleo 2.6.
- Son como los *read/write spin locks* que dan prioridad a los escritores de forma de evitar el problema de posposición indefinida para los escritores.
- Los escritores incrementan en uno al comenzar y finalizar.
- Los lectores leen al comienzo y al final hasta que le de el mismo número par.
- Los escritores van y acceden directamente sin esperar por los lectores.
- Es una buena técnica si la cantidad de escritores es realmente baja en comparación a los lectores.
- No se pueden aplicar cuando se utilizan punteros en los datos.

Seqlocks

- Para inicializar un seqlock

```
seqlock_t mr_seq_lock = DEFINE_SEQLOCK(mr_seq_lock);
```

- Para obtener un lock de write:

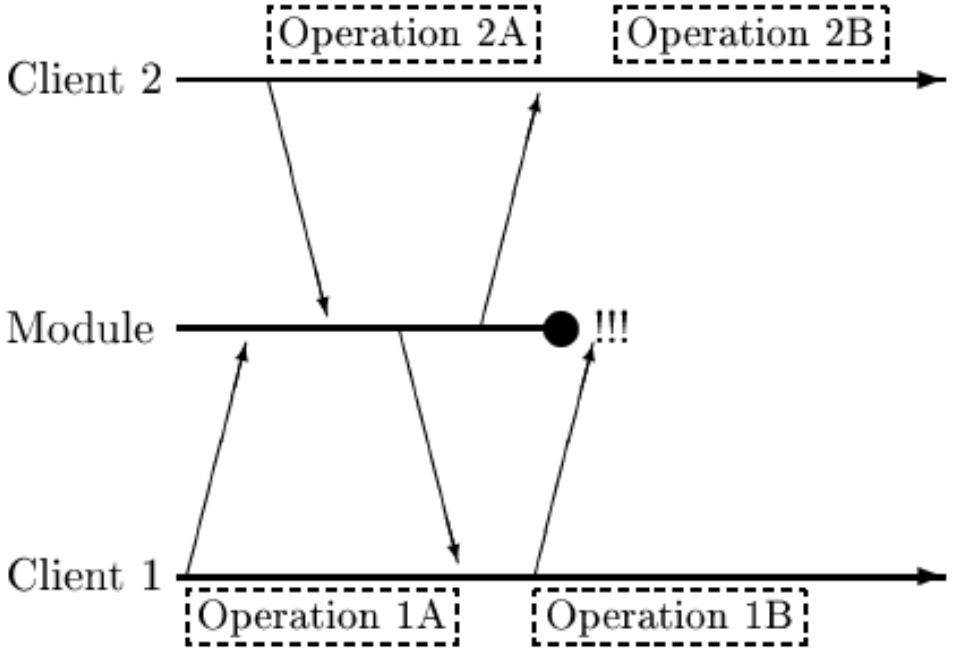
```
write_seqlock(&mr_seq_lock);
/* se obtuvo un write lock... */
write_sequnlock(&mr_seq_lock);
```

- Para realizar una lectura:

```
do {
seq = read_seqbegin(&mr_seq_lock);
/* aquí se leen los datos ... */
} while (read_seqretry(&mr_seq_lock, seq));
```

Read-Copy Update

- La utilización de esquemas *locks* de protección ante infrecuentes situaciones genera *overhead* y reducen la escalabilidad en sistemas multiprocesadores.
- Cuando las escrituras son infrecuentes, es posible utilizar un esquema de sincronización sin utilizar *locks.*

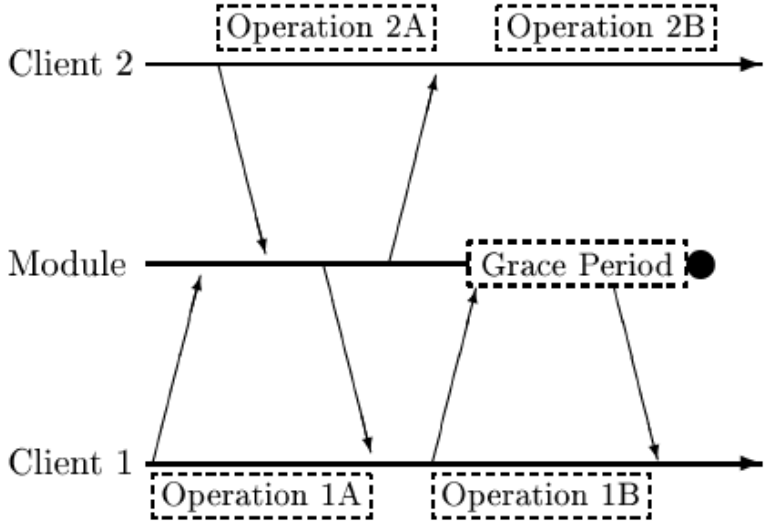


Read-Copy Update

- Los lectores acceden a la estructura de datos directamente.
- Los escritores primero generan una copia de la estructura, operan sobre la copia y luego realizan una modificación atómica de los datos.
- El problema son los lectores que permanecen con la copia vieja.
- Se les brinda un período de gracia hasta que todos terminen con su tarea.
- Una vez que expira el período se realiza la tarea de limpieza.

Read-Copy Update

- El final del período de gracia es detectado en forma indirecta cuando cada procesador pasa por un estado *"quiescent state"*:
  - *El procesador realiza un cambio de contexto*
  - *El procesador comienza a ejecutar en modo usuario*
  - *El procesador comienza a ejecutar el idle loop*



Read-Copy Update

- Las limitaciones que tienen son:
  - *Solamente estructuras de datos que sean asignadas en forma dinámica y referenciadas por punteros pueden ser protegidas por RCU.*
  - *Ningún kernel control path puede dormir dentro de una región crítica protegida por una RCU.*

Semáforos

- Es una técnica de *lock* que permite a los que esperan suspender su ejecución liberando el procesador.
- La ejecución es reanudada cuando se obtiene el *lock*
- La estructura que representa los semáforos en Linux es `struct semaphore` y es definida en `include/asm/semaphore.h`

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
};
```

Semáforos

- `count`
  - *Si el valor es más grande que 0, el recurso está libre.*
  - *Si es 0, el recurso está siendo ocupado, pero no hay nadie esperando por él.*
  - *Si es menor que 0, el recurso no está disponible y existe al menos un proceso esperando.*
- `wait`
  - *La lista de espera de los procesos que esperan por el recurso.*
- `sleepers`
  - *Bandera que es 0 si no hay ningún proceso durmiendo y 1 sino.*

Semáforos

- Las operaciones para operar sobre semáforos son:
  - `up` *Libera el lock: Incrementa en uno el campo count y verifica su nuevo valor. Si el valor es mayor que 0, no se hace más nada. Si el valor es 0, entonces se invoca a una función que despierta a alguno de los que esperan por el recurso.*
  - `down` *Adquiere el lock: Decrementa en uno el campo count y verifica si su nuevo valor es negativo. Si es así, se invoca a una función para agregarse a la lista de procesos en espera por el lock e invoca al planificador. Si no es así, continua su ejecución.*
- La operación `down` pone al proceso en estado `TASK_UNINTERRUPTIBLE`
- Se provee de otra operación `down_interruptible` que permite a los procesos recibir señales

Semáforos

- Conclusiones sobre el uso de semáforos:
  - *El acceso a la región crítica es por un período prolongado.*
  - *Debido a que se puede suspender la ejecución de la tarea, los semáforos pueden ser utilizados solamente en el contexto de un proceso. No se pueden usar en las rutinas de atención de las interrupciones.*
  - *No se podrá adquirir un semáforo mientras se tiene un spin lock ya que se puede bloquear.*

Semáforos de lectura/escritura

- Linux brinda semáforos de lectura/escritura que permiten una mayor concurrencia en el sistema.
- La implementación se realiza a través de una cola FIFO.
- De esa forma:
  - *Mientras un proceso de escritura accede al lock, todos lo que están en la cola esperan.*
  - *Cuando se libera un escritor libera el lock y el primer elemento de la lista es un lector, se hacen ingresar todos los lectores que sigan hasta el primer escritor.*
- La estructura para los semáforos de lectura/escritura es `struct rw_semaphore`.

Semáforos de lectura/escritura

- Las operaciones son:
  - *down_read*
  - *down_write*
  - *up_read*
  - *up_write*

Resumen

Técnica	Descripción	Alcance
Variables por CPU	Duplicar estructura de datos en distintas CPUs	Todas las CPUs
Operaciones atómicas	Instrucciones read-modify-write atómicas	Todas las CPUs
Barreras de memoria	Evitar reordenamiento de instrucciones	CPU Local
Spin lock	Locking con busy wait	Todas las CPUs
Semáforos	Locking espera (sleep)	Todas las CPUs
Seqlocks	Locking basado en contador	Todas las CPUs
Read-copy-update (RCU)	Acceso sin locking a datos compartidos mediante punteros.	Todas las CPUs