



Métodos de Sincronização do Kernel

Linux Kernel Development Second Edition
By Robert Love

Tiago Souza Azevedo



Operações Atômicas

- Operações atômicas são instruções que executam atomicamente sem interrupção.
- Não é possível executar duas operações atômicas na mesma variável concorrentemente.
- O kernel provê dois conjuntos de interfaces para operações atômicas:
 - Operações sobre inteiros;
 - Operações em bits individuais.
- Essas interfaces são implementadas em cada arquitetura suportada pelo Linux. A maioria das arquiteturas ou suportam diretamente operações atômicas ou a implementam através de operações de “lock” (assegurando que as operações não ocorram simultaneamente).





Operações Atômicas em Inteiros

- Os métodos de operação atômica em inteiros operam em um tipo especial de dados *atomic_t*, ao invés de um tipo int em C.
- Este tipo é usado porque assegura que o compilador não otimize erroneamente o acesso a variável. Também torna transparente ao programador as diferenças de implementação devido as arquiteturas diferentes.
- As declarações necessárias a utilização das operações atômicas em inteiros estão em `<asm/atomic.h>`.
- As operações atômicas em inteiros são tipicamente implementadas como funções *inline*.



Exemplo de utilização

Defining an `atomic_t` is done in the usual manner. Optionally, you can set it to an initial value:

```
atomic_t v;                /* define v */
atomic_t u = ATOMIC_INIT(0); /* define u and initialize it to zero */
```

Operations are all simple:

```
atomic_set(&v, 4);        /* v = 4 (atomically) */
atomic_add(2, &v);       /* v = v + 2 = 6 (atomically) */
atomic_inc(&v);          /* v = v + 1 = 7 (atomically) */
```

If you ever need to convert an `atomic_t` to an int, use `atomic_read()`:

```
printf("%d\n", atomic_read(&v)); /* will print "7" */
```





Lista de Operações Atômicas em Inteiros

Atomic Bitwise Operation	Description
<code>void set_bit(int nr, void *addr)</code>	Atomically set the <i>nr</i> -th bit starting from <i>addr</i>
<code>void clear_bit(int nr, void *addr)</code>	Atomically clear the <i>nr</i> -th bit starting from <i>addr</i>
<code>void change_bit(int nr, void *addr)</code>	Atomically flip the value of the <i>nr</i> -th bit starting from <i>addr</i>
<code>int test_and_set_bit(int nr, void *addr)</code>	Atomically set the <i>nr</i> -th bit starting from <i>addr</i> and return the previous value
<code>int test_and_clear_bit(int nr, void *addr)</code>	Atomically clear the <i>nr</i> -th bit starting from <i>addr</i> and return the previous value
<code>int test_and_change_bit(int nr, void *addr)</code>	Atomically flip the <i>nr</i> -th bit starting from <i>addr</i> and return the previous value
<code>int test_bit(int nr, void *addr)</code>	Atomically return the value of the <i>nr</i> -th bit starting from <i>addr</i>



Operações Atômicas em Bits

- São semelhantes as operações atômicas com inteiros mas operam a nível de bit. São específicas para cada arquitetura e estão definidas em `<asm/bitops.h>`.
- Operam em endereços de memória genéricos, ou seja, diferente das operações atômicas com inteiros não define um novo tipo, podendo ser utilizado com um ponteiro para qualquer tipo desejado.





Exemplo de utilização

```
unsigned long word = 0;

set_bit(0, &word);      /* bit zero is now set (atomically) */
set_bit(1, &word);      /* bit one is now set (atomically) */
printf("%ul\n", word); /* will print "3" */
clear_bit(1, &word);    /* bit one is now unset (atomically) */
change_bit(0, &word);   /* bit zero is flipped; now it is unset (atomically) */

/* atomically sets bit zero and returns the previous value (zero) */
if (test_and_set_bit(0, &word)) {
    /* never true */
}

/* the following is legal; you can mix atomic bit instructions with normal C */
word = 7;
```



Lista de Operações Atômicas em Bits

Atomic Bitwise Operation	Description
<code>void set_bit(int nr, void *addr)</code>	Atomically set the <i>nr</i> -th bit starting from <i>addr</i>
<code>void clear_bit(int nr, void *addr)</code>	Atomically clear the <i>nr</i> -th bit starting from <i>addr</i>
<code>void change_bit(int nr, void *addr)</code>	Atomically flip the value of the <i>nr</i> -th bit starting from <i>addr</i>
<code>int test_and_set_bit(int nr, void *addr)</code>	Atomically set the <i>nr</i> -th bit starting from <i>addr</i> and return the previous value
<code>int test_and_clear_bit(int nr, void *addr)</code>	Atomically clear the <i>nr</i> -th bit starting from <i>addr</i> and return the previous value
<code>int test_and_change_bit(int nr, void *addr)</code>	Atomically flip the <i>nr</i> -th bit starting from <i>addr</i> and return the previous value
<code>int test_bit(int nr, void *addr)</code>	Atomically return the value of the <i>nr</i> -th bit starting from <i>addr</i>





Spin Locks

- Dentro de uma região crítica, podemos executar várias funções, por exemplo, se compartilharmos uma estrutura, várias ações sobre essa estrutura devem ocorrer atomicamente.
- Um "spin lock" é um bloqueio que pode ser executado por mais de uma thread de execução. Se uma thread tenta obter o "spin lock" enquanto este está fechado (já bloqueado), a thread fica ocupada em um loop esperando o "spin lock" se tornar disponível. Se o "spin lock" não está bloqueado, a thread pode imediatamente obter o "spin lock" e continuar sua execução. O "spin lock" previne que mais de uma thread entre na região crítica por vez.
- O comportamento do "spin lock" gasta tempo de processamento enquanto espera o recurso ficar disponível. Por isso o "spin lock" é útil para tarefas de pequena duração. Uma alternativa seria colocar o processo no estado sleep quando este tentasse obter um recurso já ocupado e acordá-lo quando o recurso for liberado.
- Esta técnica que será discutida adiante, implica em um overhead devido as trocas de contexto.



Spin Locks

- "Spin locks" são dependentes da arquitetura e são implementados em código assembly. O código dependente da arquitetura está definido em `<asm/spinlock.h>`. As interfaces atuais estão definidas em `<linux/spinlock.h>`. Exemplo:



```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;

spin_lock(&mr_lock);
/* critical region */
spin_unlock(&mr_lock);
```





Spin Locks

- **Cuidado a ser tomado: “Spin Locks” não são recursivos!**
- Um “spin locks” em linux não é recursivo, ou seja, se um processo tentar obter um recurso já obtido por ele através de outra chamada “spin lock”, ele ficará preso no loop esperando que ele mesmo libere o recurso, gerando um deadlock.



Spin Locks

- “Spin locks” podem ser usados com *interrupt handlers*. Um semáforo não seria útil pois o processo pode entrar no estado *sleep*. Se um “spin lock” é utilizado em um *interrupt handler*, devemos desabilitar interrupções locais (chamadas de interrupção no processador corrente) antes de obter o “spin lock”. Caso contrário é possível para uma outra *interrupção*, interromper o *interrupt handler* que já havia obtido o “spin lock”.
- O kernel provê uma interface que desabilita as interrupções e obtêm o “spin lock”. Exemplo:

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;

spin_lock_irqsave(&mr_lock, flags);
/* critical region ... */
spin_unlock_irqrestore(&mr_lock, flags);
```





Usando Spin Locks

Method	Description
<code>spin_lock()</code>	Acquires given lock
<code>spin_lock_irq()</code>	Disables local interrupts and acquires given lock
<code>spin_lock_irqsave()</code>	Saves current state of local interrupts, disables local interrupts, and acquires given lock
<code>spin_unlock()</code>	Releases given lock
<code>spin_unlock_irq()</code>	Releases given lock and enables local interrupts
<code>spin_unlock_irqrestore()</code>	Releases given lock and restores local interrupts to given previous state
<code>spin_lock_init()</code>	Dynamically initializes given <code>spinlock_t</code>
<code>spin_trylock()</code>	Tries to acquire given lock; if unavailable, returns nonzero
<code>spin_is_locked()</code>	Returns nonzero if the given lock is currently acquired, otherwise it returns zero



Semáforos

- Semáforos em Linux são “sleeping locks”. Quando uma tarefa tenta obter um semáforo já em uso, o semáforo põe a tarefa na fila de espera e a põe no estado sleep, deixando o processador livre para executar outro código. Quando o processo liberar o semáforo, uma das tarefas da fila de espera é acordada e pode obter o semáforo.
- Semáforos foram formalizadas por *Edsger Wybe Dijkstra* em 1968 como um mecanismo generalizado de bloqueio. O semáforo provê duas operações atômicas, `down()` e `up()`. A operação `down()` é usada para obter o semáforo decrementando seu contador. Se o contador for zero ou maior, o semáforo é obtido e a tarefa pode entrar na região crítica.
- Se o contador se tornar negativo, a tarefa é colocada na fila de espera. A operação `up()` é usada para liberar o semáforo após a tarefa completar a execução da região crítica. Isto é feito incrementando o contador; e se a fila de espera está vazia, uma das tarefas em espera é acordada e pode obter o semáforo.





Semáforos

- Diferente de “spin locks”, semáforos não desabilitam a preempção de kernel e, portanto uma tarefa que obtém um semáforo pode ser “preempted”.
- Um semáforo pode suportar um número arbitrário de “lock holders”. Um “spin locks” permite no máximo uma tarefa por vez, enquanto que o número de tarefas simultâneas em semáforos é escolhido pelo programador. Caso seja permitido apenas uma tarefa por vez este semáforo é chamado de semáforo binário ou *mutex*.



Criando e Inicializando Semáforos

- A implementação de semáforos é dependente da arquitetura e definido em `<asm/semaphore.h>`.
- Semáforos são criados estaticamente via

```
DECLARE_SEMAPHORE_GENERIC(name, count)
```

onde “*name*” é o nome da variável e “*count*” é o contador de uso do semáforo. Para criar um mutex estaticamente, usa-se:

```
DECLARE_MUTEX(name);
```

- Mais freqüentemente, semáforos são criados dinamicamente. Neste caso, para inicializar um semáforo criado dinamicamente usa-se:

```
sema_init(sem, count);
```

onde “*sem*” é um ponteiro e “*count*” é o contador.





Usando Semáforos

- A função `down_interruptible()` tenta obter o semáforo. Se esta falha, a tarefa entra no estado `TASK_INTERRUPTIBLE`. Se a tarefa recebe um sinal enquanto espera pelo semáforo, ela é “acordada” e a função `down_interruptible()` retorna **-EINTR**.
- Já a função `down()` coloca a tarefa no estado `TASK_UNINTERRUPTIBLE`, neste estado a tarefa não responde a nenhum sinal, apenas quando o semáforo é liberado.
- A função `down_trylock()` tenta obter o semáforo, se ele já estiver bloqueado a função retorna imediatamente retornando um valor diferente de zero. Caso contrário ele retorna zero e o semáforo é obtido.



Usando Semáforos

- Para liberar um semáforo, devemos chamar a função `up()`. Por exemplo:

```
/* define and declare a semaphore, named mr_sem, with a count of one */
static DECLARE_MUTEX(mr_sem);

/* attempt to acquire the semaphore ... */
if (down_interruptible(&mr_sem)) {
    /* signal received, semaphore not acquired ... */
}

/* critical region ... */

/* release the given semaphore */
up(&mr_sem);
```





Usando Semáforos

Method	Description
<code>sema_init(struct semaphore *, int)</code>	Initializes the dynamically created semaphore to the given count
<code>init_MUTEX(struct semaphore *)</code>	Initializes the dynamically created semaphore with a count of one
<code>init_MUTEX_LOCKED(struct semaphore *)</code>	Initializes the dynamically created semaphore with a count of zero (so it is initially locked)
<code>down_interruptible(struct semaphore *)</code>	Tries to acquire the given semaphore and enter interruptible sleep if it is contended
<code>down(struct semaphore *)</code>	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
<code>down_trylock(struct semaphore *)</code>	Tries to acquire the given semaphore and immediately return nonzero if it is contended
<code>up(struct semaphore *)</code>	Releases the given semaphore and wakes a waiting task, if any



Variáveis de Conclusão

- Variáveis de conclusão são um método fácil de sincronização entre duas tarefas. Uma tarefa envia um sinal para avisar que um evento ocorreu. A outra tarefa espera essas variáveis de conclusão.
- Por exemplo, a função `vfork()` (system call) usa variáveis de conclusão para “acordar” o processo pai quando os processos filhos terminam sua execução.
- Variáveis de conclusão são representadas pela estrutura “completion type”, definida em `<linux/completion.h>`. A variável conclusão são criadas e inicializadas estaticamente por:

```
DECLARE_COMPLETION(mr_comp);
```





Usando Variáveis de Conclusão

- Uma variável de conclusão é criada e inicializada dinamicamente pela função `init_completion()`.
- Em uma dada variável de conclusão, a tarefa que deseja esperar chama a função `wait_for_completion()`. Após a execução de um evento, a função `complete()` envia sinais para todas as tarefas esperando este evento.



Variáveis de Conclusão - Funções

Method	Description
<code>init_completion(struct completion *)</code>	Initializes the given dynamically created completion variable
<code>wait_for_completion(struct completion *)</code>	Waits for the given completion variable to be signaled
<code>complete(struct completion *)</code>	Signals any waiting tasks to wake up





Seq Locks

- “Seq lock” é um novo tipo de bloqueio introduzido no kernel 2.6. Este provê um mecanismo muito simples para leitura e escrita de dados compartilhados.
- Ele funciona mantendo um contador de seqüência. Quando os dados em questão devem ser escritos, um bloqueio é obtido e o contador de seqüência é incrementado. Os dados são finalmente escritos e o contador de seqüência é incrementado ao liberar o bloqueio.
- Para a leitura dos dados, o contador de seqüência é lido. Se os valores são pares então uma escrita não está sob execução (o contador começa de zero) e o leitor entra em loop.



Usando Seq Locks

- “Seq locks” são úteis para prover um bloqueio leve e escalável para uso com muitos leitores e poucos escritores.
- O bloqueio de escrita sempre consegue obter o bloqueio desde que nenhum outro escritor bloqueando a escrita.
- Leitores não afetam o bloqueio de escrita. Além disso, escritores pendentes mantêm os leitores em loop, até que todos os escritores tenham terminado.





Usando Seq Locks

To define a seq lock:

```
seqlock_t mr_seq_lock = SEQLOCK_UNLOCKED;
```

The write path is then

```
write_seqlock(&mr_seq_lock);  
/* write lock is obtained... */  
write_sequnlock(&mr_seq_lock);
```

This looks like normal spin lock code. The address comes in with the read path, which is quite a bit different:

```
unsigned long seq;  
  
do {  
    seq = read_seqbegin(&mr_seq_lock);  
    /* read data here ... */  
} while (read_seqretry(&mr_seq_lock, seq));
```



Desabilitando Preempção

- Como o kernel é preemptivo, um processo no kernel pode ser parado a qualquer instante para permitir que um processo de prioridade maior execute.
- Desta forma uma tarefa pode começar a rodar na mesma região crítica que uma outra tarefa que foi retirada da execução.
- Para prevenir isto, o código de preempção do kernel usa “spin locks” como marcadores de regiões não preemptivas. Ao obter um “spin lock”, a preempção do kernel é desativada.
- Isto também pode ser feito através da função *preempt_disable()*. A correspondente *preempt_enable()* reabilita a preempção.





Exemplo

```
preempt_disable();  
/* preemption is disabled ... */  
preempt_enable();
```

Function	Description
<code>preempt_disable()</code>	Disables kernel preemption by incrementing the preemption counter
<code>preempt_enable()</code>	Decrement the preemption counter and check and service any pending reschedules if the count is now zero
<code>preempt_enable_no_resched()</code>	Enables kernel preemption but do not check for any pending reschedules
<code>preempt_count()</code>	Returns the preemption count



Barreiras

- Ao lidar com sincronização entre múltiplos processadores ou com dispositivos de hardware, desejamos efetuar leituras e escritas em uma ordem específica.
- Em sistemas SMP (symmetrical multiprocessing systems), pode ser importante que as escritas apareçam na ordem do código, o que não é sempre garantido uma vez que o compilador pode reordenar as instruções por motivos de performance.
- Com o objetivo de garantir essa sincronização existem instruções que asseguram a ordem de execução. Estas instruções são chamadas *barreiras*.





Usando Barreiras

- A função *rmb()* provê uma barreira para leituras à memória. Esta assegura que nenhuma leitura seja reordenada através da chamada *rmb()*. Isto é, nenhuma leitura antes da chamada será reordenada para depois da chamada e vice-versa.
- A função *wmb()* provê uma barreira para escritas. Esta funciona da mesma maneira que a função *rmb()*, assegurando que escritas não sejam reordenadas através da barreira.
- A função *mb()* provê uma barreira para leituras e escritas.



Usando Barreiras

Thread 1

```
a = 3;  
rmb();  
b = 4;  
-  
-
```

Thread 2

```
-  
-  
c = b;  
rmb();  
d = a;
```





Usando Barreiras

- Vamos considerar um exemplo usando *mb()* e *rmb()*. Os valores iniciais de 'a' e 'b' são 1 e 2 respectivamente.
- Sem o uso de barreiras, em algum processador é possível que 'c' receba o novo valor de 'b', enquanto que 'd' receba o valor antigo de 'a'. Por exemplo, 'c' poderia ser igual 2 e 'd' igual a 1.
- Usando a função *mb()* asseguramos que 'a' e 'b' são escritos na ordem pretendida, enquanto que a função *rmb()* assegura que 'c' e 'd' sejam lidos na ordem.
- Como visto anteriormente, este re-ordenamento ocorre porque processadores modernos despacham instruções fora de ordem, para otimizar o uso de pipelines. As funções *rmb()* e *wmb()* informam ao processador para executar uma leitura ou escrita pendentes antes de continuar.

