PRIMITIVAS DE SINCRONIZACIÓN: **FUTEX**

Sistemas Operativos (75.08 / 95.03) - *Curso Méndez/Simó*
Facultad de Ingeniería, Universidad de Buenos Aires

Contenidos

- **Lock.** ¿Qué son? Interfaz.
- **Spinlock**. Implementación.
- **Mutex**. Primera idea de implementación.
- Syscall **futex**. ¿Por qué es necesaria?
- **Implementando un mutex** con futex(2). Optimizaciones.
- ¿Mutex o spinlock? **Spinning en user-space**.

Lock

¿Qué es?

Un **lock** es una **primitiva de sincronización** que permite **coordinar el acceso a regiones de memoria compartidas en un sistema concurrente**, garantizando que el estado final de dichas regiones sea *el esperado*.

Está diseñado para garantizar la **exclusión mutua**, esto es, evitar que más de un proceso a la vez entre en una **sección crítica** (fragmento de código donde puede modificarse la región de memoria compartida).

Tiene *(generalmente)* dos estados: **bloqueado** y **desbloqueado**.

¿Cuál es la interfaz de un lock?

La interfaz más simple de un lock está compuesta generalmente de tres métodos:

- **init():** inicializa la estructura.
- **lock():** se intenta *tomar* (bloquear) el lock. En caso de ya estar bloqueado, el thread se bloquea *(el cómo dependerá de cada primitiva)* hasta poder tomarlo.
- **unlock():** desbloquea la estructura.

Cada proceso que quiera acceder a los datos protegidos por el lock, debe llamar a su método `lock()` antes de hacerlo, y al finalizar la sección crítica, debe liberarlo utilizando el método `unlock()`.

Contención de un lock

Se dice que un lock es *contenido* cuando el thread que intenta tomarlo **debe esperar a que este sea liberado** antes de poder continuar. Por el contrario, hablaremos de locks *no contenidos* cuando el método de bloqueo inmediatamente de forma exitosa.

El **nivel de contención** de un determinado *benchmark* es un factor importante a tener en cuenta, ya que según este (y según otros factores) convendrá optar por una u otra implementación de locks.

Spinlock

¿Qué es? ¿Cómo funciona?

Es una implementación de **lock**, y como tal, permite **coordinar el acceso a un recurso compartido entre threads**, garantizando el correcto estado final mismo. A su vez, tiene dos estados posibles: **bloqueado** y **desbloqueado**.

Cuando un thread intenta adquirirlo, **entra en un ciclo** *(en el que no realiza ninguna operación)* hasta que el mismo está disponible.

Concepto de **busy-waiting:** el thread permanece activo pero sin realizar ninguna tarea útil.

No se requiere de syscalls: funcionan en **user-space**.

¿Cómo se implementa?

Una posible implementación consiste en utilizar un entero como **variable de estado**: si toma el valor 0 el lock está libre, mientras que si toma valor distinto de 0 está bloqueado.

Con este enfoque, para adquirir el lock deberíamos verificar que el mismo esté libre *(en este caso, que el estado sea 0)* y marcarlo como tomado *(el estado pasa a ser 1)*. Para liberar el lock, basta igualar el estado a 0.

Primera idea

```
void acquire(struct lock *lk) {
    while(lk->flag);
    lk->flag = 1;
}

void release(struct lock *lk) {
    lk->flag = 0;
}
```

¿Funciona? ¿Qué problemas puede traer esta implementación?

¡Atomicidad!

Si bien parece sencillo, es fundamental lograr que la adquisición y liberación del mismo se realice **atómicamente**, de lo contrario podría producirse un *context switch* justo después de salir del *while*, permitiendo a otro thread bloquear el spinlock al mismo tiempo.

Necesitamos entonces de alguna **instrucción especial** que nos permita **manejar datos de forma atómica**.

Implementación en x86

Suponiendo arquitectura x86, podemos implementar el spinlock utilizando la instrucción **XCHG**, que nos permite hacer lo que se conoce como **test-and-set**: escribir un valor (en este caso 1) en una dirección de memoria, retornando el antiguo valor de forma atómica.

```
static inline int xchg(volatile int *addr, int newval) {
    int result;

    __asm__ volatile("xchgl %0, %1"
                 : "+m"(*addr), "=a"(result)
                 : "1"(newval)
                 : "cc");

    return result;
}

static inline void clear(volatile int *addr) {
    __asm__ volatile("movl $0, %0" : "+m"(*addr) :);
}

void acquire(struct lock *lk) {
    while(xchg(&lk->flag, 1));
}

void release(struct lock *lk) {
    clear(&lk->flag);
}
```

Implementación genérica

Por portabilidad, no podemos depender de instrucciones específicas de una arquitectura. Para implementar spinlocks genéricos, necesitamos que el lenguaje exponga **primitivas atómicas de manejo de datos**.

El estándar C11 introdujo soporte para tipos atómicos en C.

```c
#include <stdatomic.h>

void acquire(struct lock *lk) {
    while(atomic_flag_test_and_set(&lk->flag));
}

void release(struct lock *lk) {
    atomic_flag_clear(&lk->flag);
}
```

Mutex

¿Qué es?

Al igual que el spinlock, el mutex *(**mut**ual **ex**clusion)* es otra primitiva que implementa la interfaz de un **lock**. Como tal, tiene dos estados posibles: **bloqueado** y **desbloqueado**.

¿Cómo funciona?

Cuando un thread quiere adquirirlo y el mismo se encuentra tomado, el thread solicitante se pone a “dormir”, a la vez que entra en una **cola de espera** *(“waiters”)*. Cuando el mutex es liberado por otro thread, se **despierta a los “waiters”** para que vuelvan a **disputarse** el acceso.

Sin embargo, para poner un thread a “dormir” y despertarlo con una señal posterior, es necesario soporte del lado del **kernel**.

Para adquirir y liberar un mutex entonces es necesario utilizar **syscalls**, y es importante notar que las syscalls no son gratis.

Implementación *naive* en JOS

Basándose en el spinlock, podríamos en una primer iteración proponer la siguiente implementación:

```
void lock(struct mutex *m) {
    while(atomic_flag_test_and_set(&m->flag))
        sys_yield();
}

void unlock(struct mutex *m) {
    atomic_flag_clear(&m->flag);
}
```

La idea es **ceder el control** voluntariamente cuando el mutex está bloqueado para esperar que se libere. Esta idea, ¿funciona? ¿Qué problemas puede traer?

¡Muchos *wake-ups*!

La implementación **no pone a dormir al thread solicitante**, sino que simplemente cede el control para que otros threads se ejecuten.

Si bien esto funciona ya que permite a otros threads liberar el mutex mientras el solicitante espera, es importante notar que el thread en ningún momento se marca como `ENV_NOT_RUNNABLE`, por lo que el scheduler lo va a volver a ejecutar **aun cuando el mutex no se haya liberado**.

El gran problema es que cada wake-up innecesario incluye *context switches* costosos, impactando negativamente sobre la performance.

Pero entonces... *¿cómo implementamos un mutex?*

Como vimos, para implementar un mutex es **necesario** que el **kernel** nos provea de *alguna herramienta* para “dormir” a los threads que intenten adquirir un mutex bloqueado, y despertarlos cuando el mismo se libere para que puedan disputarlo nuevamente.

futex(2)

¿Qué es?

Futex *(fast userspace mutex)* es una **syscall** introducida por primera vez en Unix en la versión 2.5.7. que **permite implementar abstracciones de sincronización de alto nivel** tales como mutexes, condition variables, barreras y semáforos.

Obs.: glibc no provee un wrapper para esta syscall, por lo que hay que llamarla utilizando syscall(2).

¿En qué consiste?

Un **futex** consiste en una **cola de espera** en *kernel-space* atada a un **entero atómico** en *user-space*.

Los distintos hilos de un sistema concurrente pueden entonces manipular este entero en user-space *(de forma atómica para evitar race conditions)* para **coordinar el acceso a una región de memoria compartida**, utilizando la syscall **futex(2)** ya sea para ponerse a sí mismos en la cola de espera o para “despertar” a otros procesos que están esperando.

Es importante notar que esto permite utilizar syscalls sólo en el caso de que un lock sea contenido, no siendo necesario en caso de que el mutex esté libre.

Operaciones principales

futex(2) provee **dos* funcionalidades básicas**:

- **WAIT(addr, val):** si el valor guardado en `addr` es igual a `val`, pone al hilo actual a dormir (en la cola de espera asociada a `*addr*`).
- **WAKE(addr, num):** despierta `num` número de hilos que están esperando en la cola de espera asociada a `addr`.

**(Existen más operaciones, ver el manual.)*

Implementando un mutex con futex(2)

*Implementaciones basadas en el paper canónico **“Futexes Are Tricky”** (2011) de Ulrich Drepper*

Funciones requeridas

Asumimos la existencia de las siguientes funciones y wrappers:

- **atomic_inc(var):** se incrementa atómicamente la variable `var` y se retorna el valor inicial.
- **atomic_dec(var):** se decrementa atómicamente la variable `var` y se retorna el valor inicial.
- **cmpxchg(var, old, new):** si el valor actual de `var` es `old`, se reemplaza con `new`. Se retorna el valor inicial de var.
- **futex_wait(futex, val):** si el valor de `futex` es `val` pone el hilo a dormir en la cola asociada a `futex`.
- **futex_wake(futex, num):** despierta hasta `num` número de hilos de la cola asociada a `futex`.

(La implementación de las mismas puede variar según el lenguaje y la arquitectura.)

Primera implementación

Nuestro mutex debe tener una variable de **estado** que permita saber si el mismo está bloqueado o no. Para esta primera implementación, nos basta con que estado valga 0 si está libre, o un valor mayor a 0 si está bloqueado.

```
typedef struct mutex {
    // 0 == unlocked
    // >0 == locked
    volatile int flag;
} mutex_t;
```

lock()

Para bloquear el mutex, se incrementa atómicamente la variable de estado, y se evalúa el valor anterior:

- Si el valor anterior era 0, se adquiere el mutex (la variable de estado vale 1).
- Si el valor no era 0, se pone el thread a dormir.

```
void lock(struct mutex *m) {
    int c;
    while ((c = atomic_inc(&m->flag)) != 0)
        futex_wait(&m->flag, c + 1);
}
```

unlock()

Para desbloquear el mutex, se escribe la variable de estado con 0 (desbloqueado), y posteriormente se despierta a hasta un *waiter.*

```
void unlock(struct mutex *m) {
    m->flag = 0;
    futex_wake(&m->flag, 1);
}
```

¿Problemas?

```
void lock(struct mutex *m) {
    int c;
    while ((c = atomic_inc(&m->flag)) != 0) futex_wait(&m->flag, c + 1);
}

void unlock(struct mutex *m) {
    m->flag = 0;
    futex_wake(&m->flag, 1);
}
```

Problema de eficiencia

El principal problema de eficiencia se ve reflejado en el método `unlock`: **siempre se hace la llamada a la syscall `futex_wake`**, incluso en el escenario en que el mutex nunca es disputado por otros hilos.

Si pudiéramos diferenciar entre *bloqueado con hilos a la espera* y *bloqueado sin hilos esperando*, podríamos evitar esta llamada cuando resulte innecesaria.

Problemas de funcionalidad

Más allá de la posible optimización a realizar en `unlock`, esta primer implementación tiene **dos errores**:

- En `lock`, entre la lectura e incremento de la variable y la llamada a `futex_wait`, existe la posibilidad de que **otro hilo quiera hacer exactamente lo mismo**, pudiendo entrando en un loop infinito. Vemos que el `lock` no resulta ser atómico.
- El segundo error está en que podría existir un ***overflow*** en la **variable de estado** (debido a que no hay límite en cuanto a la cantidad de hilos que pueden entran a la cola de *waiters* llamando a `lock`).

Nueva implementación propuesta

Ahora se propone que la variable de **estado** tenga **tres posibles valores**:

- 0 si está libre,
- 1 si está bloqueado y no hay hilos esperando que se libere,
- 2 si está bloqueado y hay al menos un hilo esperando que se libere.

```
typedef struct mutex {
    // 0 == unlocked
    // 1 == locked, no waiters
    // 2 == locked, one or more waiters
    volatile int flag;
} mutex_t;
```

lock() no contenido

El primer cambio que podemos observar, es que ahora comenzamos utilizando la función `cmpxchg` para **distinguir el caso no contenido del resto**: si el valor del mutex era 0, se lo marca con el valor 1, correspondiente a “bloqueado sin *waiters*” y la función termina.

```
void lock(struct mutex *m) {
    int c;
    if ((c = cmpxchg(&m->flag, 0, 1)) != 0)
        do {
            if (c == 2 || cmpxchg(&m->flag, 1, 2) != 0)
                futex_wait(&m->flag, 2);
        } while ((c = cmpxchg(&m->flag, 0, 2)) != 0);
}
```

lock() contenido

En cambio, si el valor anterior era 1 o 2, **el lock es contenido**, por lo que debemos esperar.

Si inicialmente no habían hilos esperando (el valor era 1), debemos **marcar que ahora hay hilos esperando**, poniendo el **valor en 2**. Para esto utilizamos el segundo `cmpxchg`. Al hacerlo, además, chequeamos que entre ambas operaciones el mutex no haya sido liberado por otro hilo. Si efectivamente lo fue, intentamos **adquirir el mutex inmediatamente** sin llamar a `futex_wait`.

```
void lock(struct mutex *m) {
    int c;
    if ((c = cmpxchg(&m->flag, 0, 1)) != 0)
        do {
            if (c == 2 || cmpxchg(&m->flag, 1, 2) != 0)
                futex_wait(&m->flag, 2);
        } while ((c = cmpxchg(&m->flag, 0, 2)) != 0);
}
```

Luego de retornar del `futex_wait` *(o de evitar la llamada)* volvemos a intentar adquirir el mutex. En este caso, **debemos obligatoriamente establecer la variable de estado en 2**, debido a que en este punto no podemos saber si hay o no otro *waiter*.

Como consecuencia, es posible que estemos realizando una llamada innecesaria a `futex_wake` en el unlock, como veremos a continuación.

```
void lock(struct mutex *m) {
    int c;
    if ((c = cmpxchg(&m->flag, 0, 1)) != 0)
        do {
            if (c == 2 || cmpxchg(&m->flag, 1, 2) != 0)
                futex_wait(&m->flag, 2);
        } while ((c = cmpxchg(&m->flag, 0, 2)) != 0);
}
```

unlock()

Ahora podemos realizar la optimización previamente mencionada.

Se añade una operación atómica con la intención de **evitar la syscall** en el caso en que **sabemos que no hay hilos esperando.**

Si el valor anterior a decrementar la variable era 2, esto significa que es posible que **hayan hilos esperando y que debemos despertarlos** (debido a la implementación del `lock`, no necesariamente los hay).

```
void unlock(struct mutex *m) {
    if (atomic_dec(&m->flag) != 1) {
        m->flag = 0;
        futex_wake(&m->flag, 1);
    }
}
```

Spinning en *user-space*

¿Mutex o spinlocks?

Como se vió, ambas primitivas implementan un efectivo **lock** para coordinar el acceso a una región de memoria compartida en un sistema concurrente. Sin embargo, **¿cuál es mejor?**

La respuesta es **depende**. No hay una primitiva “mejor” para todos los escenarios: va a depender en cada caso de la **contención**, del número de núcleos que tenga el CPU, de la arquitectura, etc.

El problema de los mutexes

Como vimos, los mutexes POSIX están implementados utilizando la **syscall futex(2)**, por lo que poner a “dormir” threads y despertarlos son **operaciones costosas** que involucran cambios de contexto y muchas instrucciones de CPU.

Si el mutex se bloquea por poco tiempo, es probable que el tiempo empleado en poner a dormir al hilo y luego despertarlo sea mucho mayor al tiempo de utilización, y al mismo tiempo mayor al tiempo que hubiese empleado un **spinlock.**

Mutexes híbridos: *spinning en user-space*

Futex permite lo que se conocen como **mutexes híbridos**: en el caso **contenido** en que un hilo no puede adquirir el mutex y debe esperar, se decide realizar una **etapa de spinning** en **user-space** (como si se tratase de un *spinlock*) en lugar de poner al hilo a dormir inmediatamente.

De esta forma, es posible evitar el *overhead* introducido por las llamadas a la **syscall futex** en caso de que el mutex se libere *pronto*.

Para seguir investigando...

Implementación de *condition variables* con futex(2)

- Condition variable implementation with Futex: https://www.remlab.net/op/futex-condvar.shtml
- Mutexes and Condition Variables using Futexes: https://locklessinc.com/articles/mutex_cv_futex/

Bibliografía

- Wikipedia: https://en.wikipedia.org/wiki/Lock_(computer_science), https://en.wikipedia.org/wiki/Mutual_exclusion, https://en.wikipedia.org/wiki/Concurrency_control
- Linux manual page (https://man7.org/linux/man-pages/man2/futex.2.html)
- U. Drepper: “Futexes Are Tricky” (2011).
- H. Franke et al.: “Fuss, Futexes and Furwocks: Fast Userlevel Locking” (2002).
- Eli Bendersky website: https://eli.thegreenplace.net/2018/basics-of-futexes
- Stack overflow: https://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex/5870415#5870415

¿Preguntas?

¡Gracias!