Sistemas Operativos

Sincronización entre procesos/threads

Pablo Ibáñez

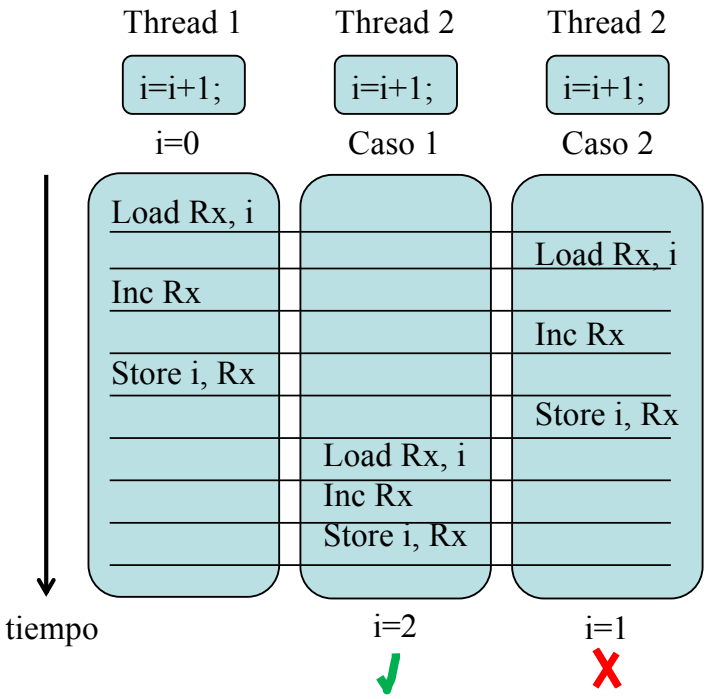
Indice

- Conceptos de exclusión mutua y de sección crítica
- Mutex, Semáforos
- Monitores y variables condición
- Implementación
- Ejemplo

[SGG]: capítulo 6
[Ste94]: capítulo 11

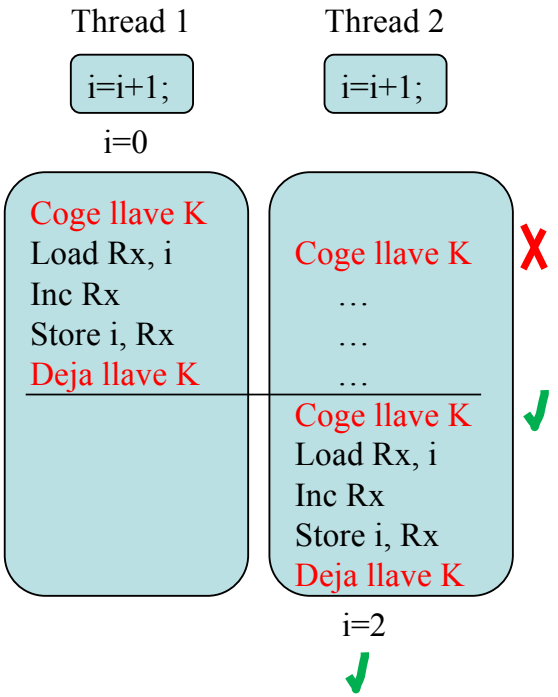
Problema: acceso concurrente a variable compartida

- Ejemplo: acceso concurrente/paralelo de dos threads a variable compartida
  - Threads 1 y 2 incrementan variable i
- Necesidad de **exclusión mutua**



Sección crítica

- Sección de código protegida por una llave
- Las secciones críticas protegidas por la misma llave se ejecutan en exclusión mutua



Herramientas de bajo nivel: mutex

- Dos operaciones:
  - Coge llave: lock, wait
  - Deja llave: unlock, signal

Thread 1

```
lock(S);
i=i+1;
unlock(S);
```

Thread 2

```
lock(S);
i=i+1;
unlock(S);
```

```
void lock(S){
  while (S!=0) esperar;
  S=1;
}
```

```
void unlock(S){
  S==0;
}
```


Herramientas de bajo nivel: semáforo

- Extensión de mutex para varias llaves
- Dos operaciones:
  - Coge llave: P, down, sleep, wait
  - Deja llave: V, up, wakeup, signal

```
void wait(S){
  while (S<=0);
  S--;
}
```

```
void signal(S){
  S++;
}
```

Implementación con espera activa

Herramientas de bajo nivel: semáforo

- Implementación sin espera activa

```
void wait(S)
{
  S.cuenta --;
  If (S.cuenta<0)
  {
    añadir_thread(S.cola);
    bloquear_thread();
  }
}
```

```
void signal(S)
{
  S.cuenta ++;
  If (S.cuenta<=0)
  {
    T=sacar_thread(S.cola);
    despertar_thread(T);
  }
}
```

Si (S.cuenta>0) indica número de threads que aún pueden cruzar el semáforo
Si (S.cuenta<0) indica número de threads encolados, esperando a cruzar semáforo

Herramientas de alto nivel: monitores

- Requiere lenguaje orientado a objetos
- Monitor
  - Objeto: variables + procedimientos
  - Algunos procedimientos se ejecutan en exclusión mutua
    - En Java → los declarados como synchronized
  - Una llave única se coge a la entrada de cada función synchronized y se deja a la salida

Herramientas de alto nivel: variables condición

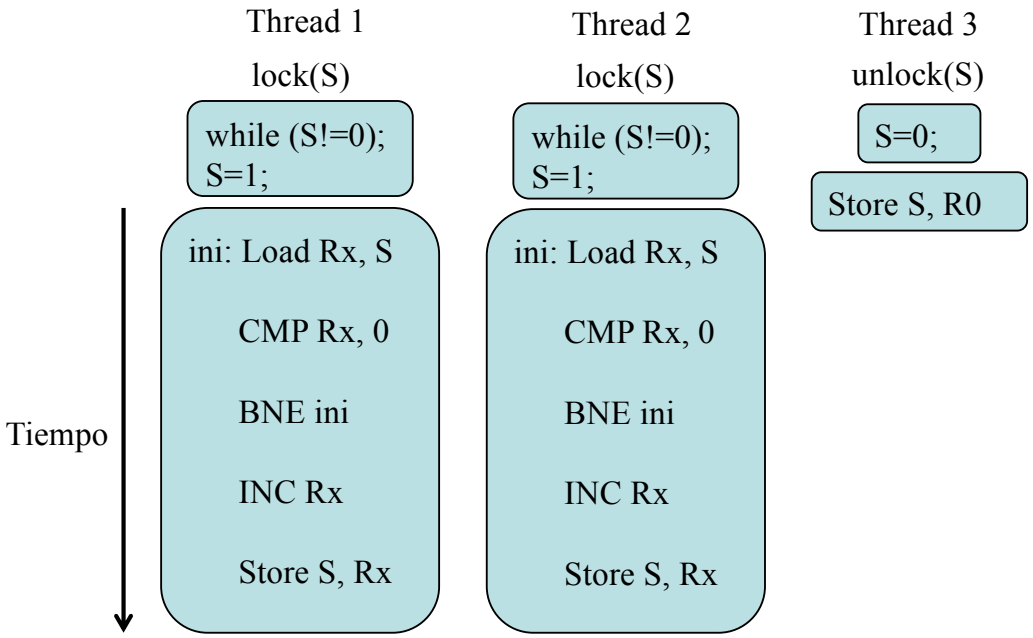
- Variable con dos operaciones
  - wait (variable), bloquea a quien la llama
  - signal (variable), desbloquea a uno de los threads bloqueados en la variable
- Pueden proporcionar otra de broadcast
  - broadcast (variable), desbloquea a todos los threads bloqueados en la variable
- Parecido a un mutex pero NO es lo mismo
  - Mutex generalmente empieza abierto, el primer proceso que ejecuta lock adquiere la llave
  - Condition variable nunca esta abierta. Cuando un thread ejecuta wait, siempre se bloquea. Permanece bloqueado hasta que otro thread ejecute signal.

Si pero… funciona ?

- Todos los mecanismos que hemos visto se basan en llaves que hay que coger para poder acceder a una zona de código y dejar cuando se abandona la zona
- Pero repasemos la operación de coger llave

```
void wait(S){
   while (S<=0);
   S--;
}
```

Si pero… funciona ?



- ERROR → Threads 1 y 2 cogen la llave

Solución: exclusión mutua

- Las instrucciones necesarias para coger llave deben ejecutarse en exclusión con otros threads
- Pero … hemos vuelto al principio !
- Necesitamos ayuda
  - Del sistema operativo
  - Del hardware

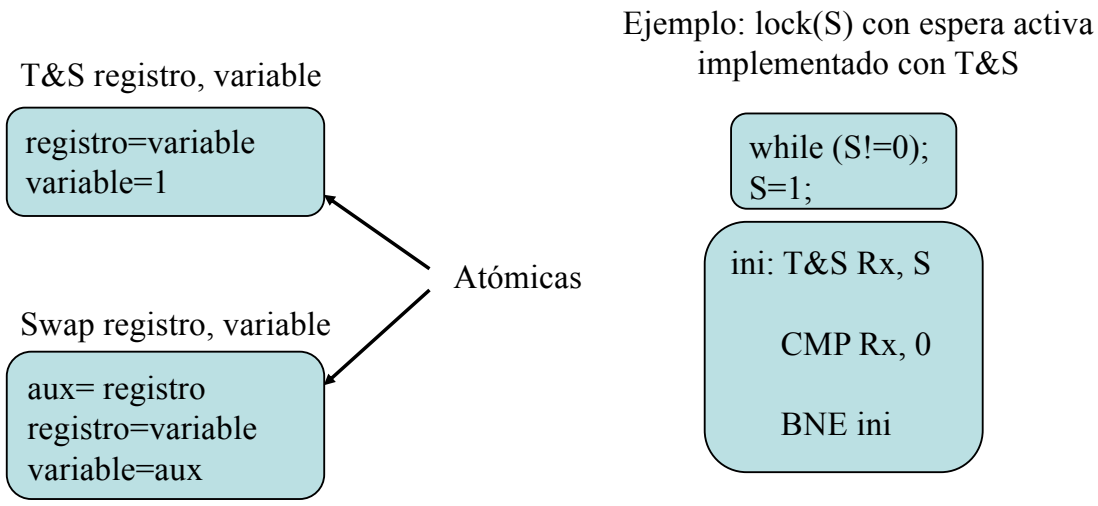
Solución sistema operativo

- Solo para sistemas mono-procesador
- Inhibir interrupciones durante la ejecución de las instrucciones de coger llave
  - Se impide el cambio de contexto
- Requiere llamada al sistema

- Alternativa:
  algoritmos de Dekker (1965) y Peterson (1981)

Solución hardware

- Instrucciones atómicas de lectura y escritura en memoria



POSIX threads

- Implementados con espera no activa. Cuando un thread no puede avanzar, se bloquea.

- Mutex
- Reader-writer locks
- Condition variables

POSIX threads: mutex

```
pthread_mutex_init
     (pthread_mutex_t *mutex, atributos)
pthread_mutex_destroy (pthread_mutex_t *mutex)
pthread_mutex_lock (pthread_mutex_t *mutex)
pthread_mutex_unlock (pthread_mutex_t *mutex)
pthread_mutex_trylock (pthread_mutex_t *mutex)
```

- Unlock() despierta a todos los threads bloqueados en un mutex. El primero en llegar entra, los demás verán el mutex cerrado y volverán a quedar bloqueados

POSIX threads: reader-writer locks

```
pthread_rwlock_init
      (pthread_rwlock_t *rwlock, atributos)
pthread_rwlock_destroy (pthread_rwlock_t *rwlock)
pthread_rwlock_rdlock (pthread_rwlock_t *rwlock)
pthread_rwlock_wrlock (pthread_rwlock_t *rwlock)
pthread_rwlock_unlock (pthread_rwlock_t *rwlock)
pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock)
pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock)
```

- Similar a mutex pero con tres estados:
  - Locked in read mode. Pueden varios threads a la vez en este modo
  - Locked in write mode. Solo un thread
  - Unlocked. No lo tiene nadie

POSIX threads: condition variables

```
pthread_cond_init
      (pthread_cond_t *cond, atributos)
pthread_cond_destroy (pthread_cond_t *cond)
pthread_cond_wait
      (pthread_cond_t *cond, pthread_mutex_t *mutex)
pthread_cond_signal (pthread_cond_t *cond)
pthread_cond_timedwait
      (.. *cond,.. *mutex, struct timespec * timeout)
pthread_cond_broadcast (pthread_cond_t *cond)
```

- El mutex pasado a wait protege la condición. Quien llama a wait debe adquirir antes el mutex. Wait encola al thread y desbloquea el lock. Cuando wait devuelve control también devuelve el mutex bloqueado

- int pthread_spin_lock(pthread_spinlock_t *lock);
- int pthread_spin_trylock(pthread_spinlock_t *lock);
- int pthread_spin_destroy(pthread_spinlock_t *lock);
- int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
- int pthread_spin_unlock(pthread_spinlock_t **lock*);

Mutex implementados con espera activa