

2º curso / 2º cuatr.
Grado en
Ing. Informática

Arquitectura de Computadores

Tema 3

Lección 10. Sincronización

Material elaborado por los profesores responsables de la asignatura:
Mancia Anguita – Julio Ortega

Licencia Creative Commons



ugr

Universidad
de Granada



Lecciones

- Lección 7. Arquitecturas TLP
- Lección 8. Coherencia del sistema de memoria
- Lección 9. Consistencia del sistema de memoria
- Lección 10. Sincronización
 - Comunicación en multiprocesadores y necesidad de usar código de sincronización
 - Soporte software y hardware para sincronización
 - Cerrojos
 - Cerrojos simples
 - Cerrojos con etiqueta
 - Barreras
 - Apoyo hardware a primitivas software

Objetivos Lección 10

- Explicar por qué es necesaria la sincronización en multiprocesadores.
- Describir las primitivas para sincronización que ofrece el hardware.
- Implementar cerros simples, cerros con etiqueta y barreras a partir de instrucciones máquina de sincronización y ordenación de accesos a memoria.

Bibliografía Lección 10

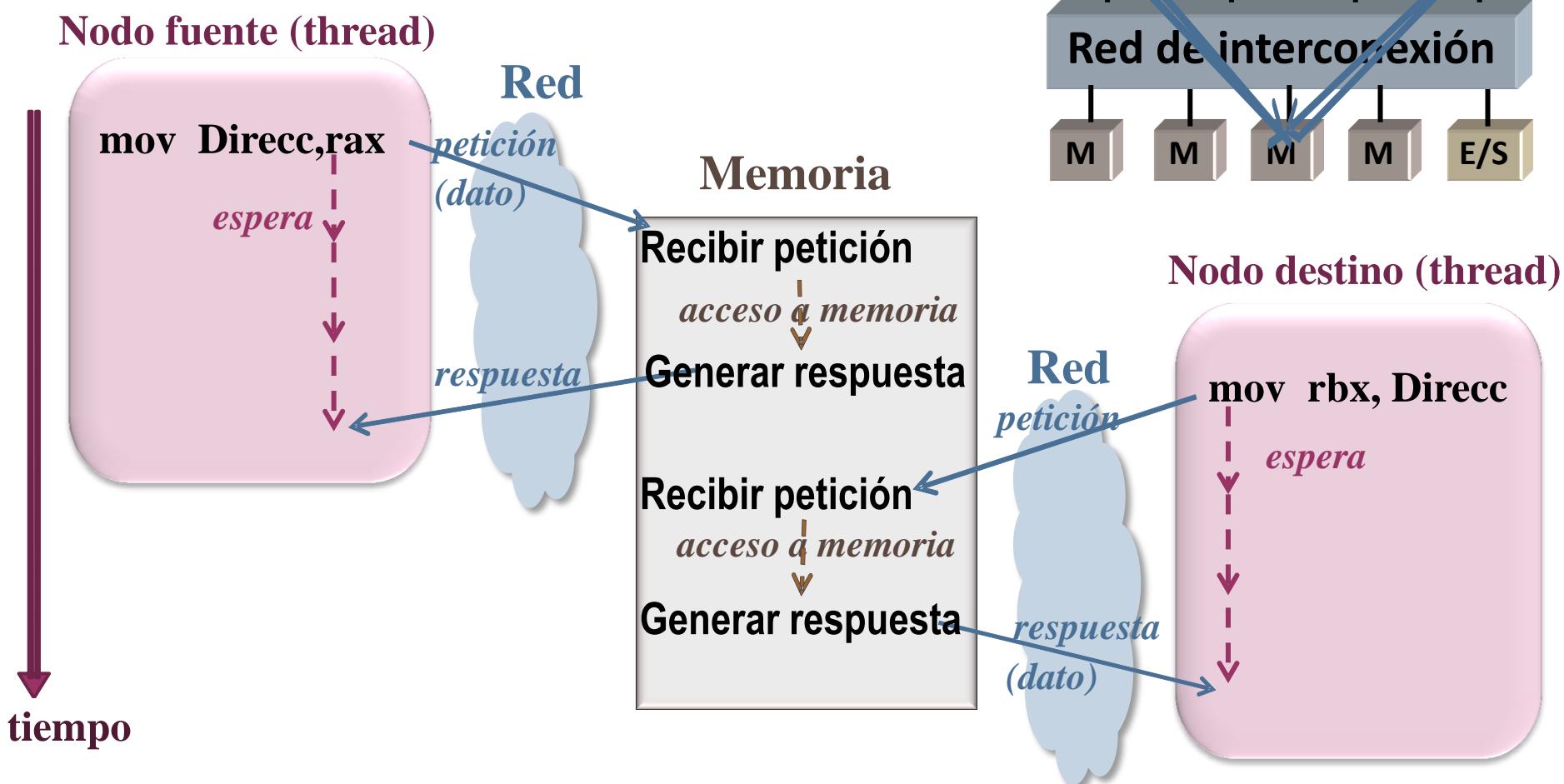
➤ Fundamental

- Cap.3, Secc. 5. M. Anguita, J. Ortega. *Fundamentos y Problemas de Arquitectura de Computadores*. Librería Fleming/Ed. Avicam, 2016.
- Secc. 10.3. J. Ortega, M. Anguita, A. Prieto. “Arquitectura de Computadores”. ESII/C.1 ORT arq

Contenido Lección 10

- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software

Comunicación en un multiprocesador



Comunicación uno-a-uno

Secuencial	Paralela	
	<u>P1</u>	<u>P2</u>
... A=valor; ... copia=A; A=valor; copia=A; ...
... mov A,rx ... mov rbx,A mov A,rx mov rbx,A ...

Comunicación uno-a-uno. Necesidad de sincronización

- Se debe garantizar que el proceso que recibe lea la variable compartida cuando el proceso que envía haya escrito en la variable el dato a enviar

- Si se reutiliza la variable para comunicación, se debe garantizar que no se envía un nuevo dato en la variable hasta que no se haya leído el anterior

Paralela (inicialmente K=0)	
P1	P2
... A =1; K =1; while (K ==0) { }; copia= A ; ...

Comunicación colectiva

Secuencial	Paralela (sum=0)
<pre>for (i=0 ; i<n ; i++) { sum = sum + a[i]; } printf(sum);</pre>	<pre>for (<i>i</i>=<i>ithread</i> ; <i>i</i>< n ; <i>i</i>=<i>i</i>+<i>nthread</i>) { <i>sump</i> = <i>sump</i> + a[<i>i</i>]; } sum = sum + <i>sump</i>; /* SC, sum compart. */ if (<i>ithread</i>==0) printf(sum);</pre>

- Ejemplo de comunicación colectiva: suma de n números:
 - La lectura-modificación-escritura de `sum` se debería hacer en exclusión mutua (es una sección crítica) => **cerros**
 - Sección crítica: Secuencia de instrucciones con una o varias direcciones compartidas (variables) que se deben acceder en exclusión mutua
 - El proceso 0 no debería imprimir hasta que no hayan acumulado `sump` en `sum` todos los procesos => **barreras**

Comunicación colectiva en multiprocesadores (carrera)

Sistema de memoria

	Ej.1	Ej.2		Ej.1	Ej.2
R ₀ (suma)	1.0	2.0	R ₂ (suma)	3.1	1.0
W ₀ (suma)	2.1	3.1	W ₂ (suma)	4.4	4.3
R ₁ (suma)	5.4	7.7	R ₃ (suma)	7.6	5.3
W ₁ (suma)	6.6	8.9	W ₃ (suma)	8.10	6.7

Thread 0 **Thread 1**

Orden.resultado

- Ej. para n=4, el compilador no optimiza
- a={1,2,3,4}
- R_i (suma): Lectura de suma en la iteración i

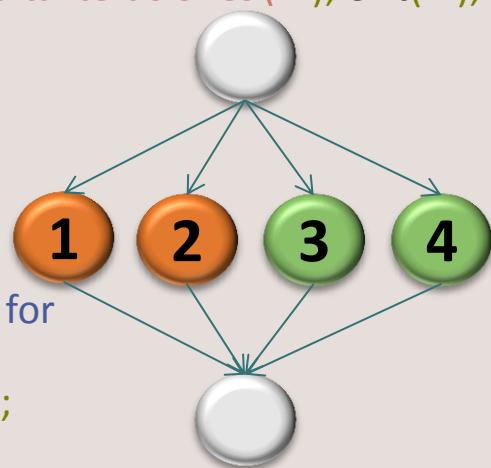
R_i(suma) ←
↓
W_i(suma) ←

sin exclusión mutua en el acceso a suma

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
main(int argc, char **argv) {
    int i, n=20, a[n],suma=0;
    if(argc < 2) {
        fprintf(stderr,"\\nFalta iteraciones\\n"); exit(-1);
    }
    n = atoi(argv[1]);
    if (n>20) n=20;
    for (i=0; i<n; i++)
        a[i] = i+1;
```

```
#pragma omp parallel for
for (i=0; i<n; i++)
    suma = suma + a[i];
printf("Fuera de 'parallel' suma=%d\\n",suma);
return(0);
```

}



Contenido Lección 10

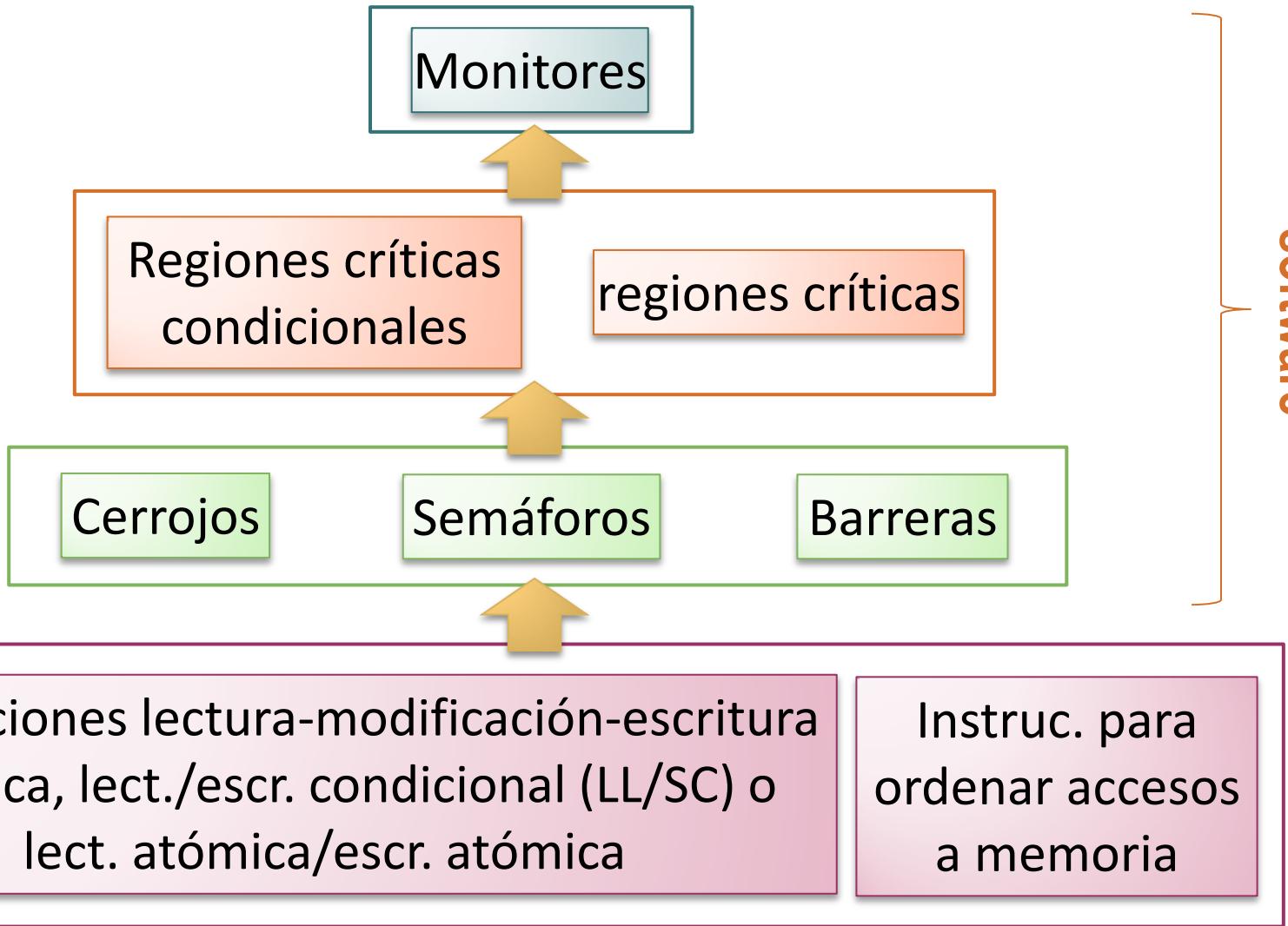
- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software

Soporte software y hardware de sincronización

abstracción

software

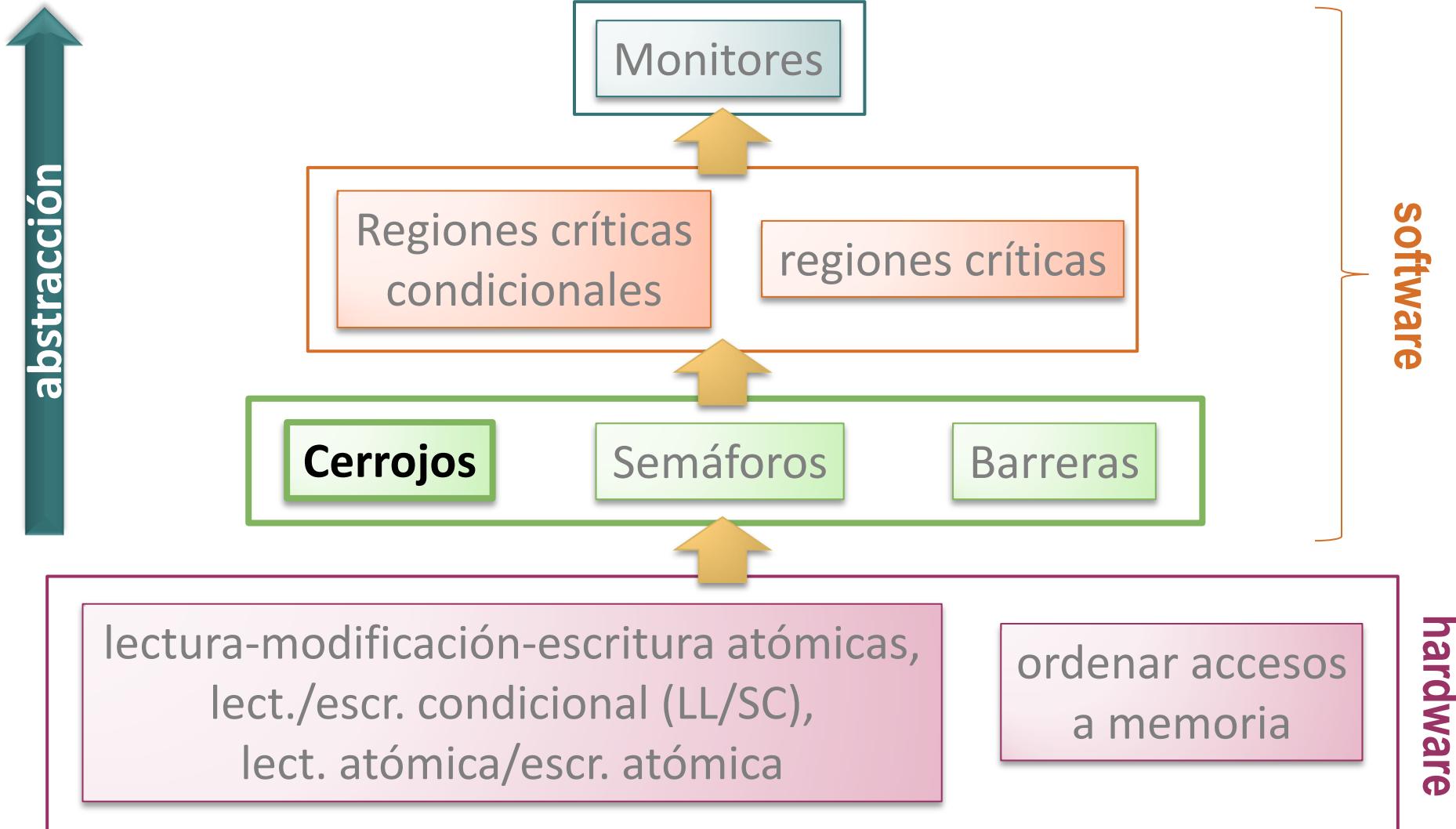
hardware



Contenido Lección 10

- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
 - Cerrojos simples
 - Cerrojos con etiqueta
- Barreras
- Apoyo hardware a primitivas software

Soporte software y hardware de sincronización



Cerrojos

- Permiten sincronizar mediante dos operaciones:
 - Cierre del cerrojo o lock (k) : intenta **adquirir** el derecho a acceder a una sección crítica (cerrando o adquiriendo el cerrojo k).
 - Si varios procesos intentan la **adquisición** (cierre) a la vez, sólo uno de ellos lo debe conseguir, el resto debe pasar a una *etapa de espera*.
 - Todos los procesos que ejecuten `lock ()` con el cerrojo cerrado deben quedar **esperando**.
 - Apertura del cerrojo o unlock (k) : **libera** a uno de los threads que esperan el acceso a una sección crítica (éste adquiere el cerrojo).
 - Si no hay threads en **espera**, permitirá que el siguiente thread que ejecute la función `lock ()` adquiera el cerrojo k sin espera

Cerrojos en ejemplo suma

Secuencial	Paralela
<pre>for (i=0 ; i<n ; i++) { sum = sum + a[i]; }</pre>	<pre>for (<i>i</i>=<i>ithread</i> ; <i>i</i><<i>n</i> ; <i>i</i>=<i>i</i>+<i>nthread</i>) { <i>sump</i> = <i>sump</i> + a[<i>i</i>]; } lock(k); sum = sum + sump; /* SC, sum compart. */ unlock(k);</pre>

- Alternativas para implementar la espera:
 - Espera ocupada.
 - Suspensión del proceso o thread, éste queda esperando en una cola, el procesador conmuta a otro proceso-thread.

Componentes en un código para sincronización

➤ Método de adquisición

- Método por el que un thread trata de adquirir el derecho a pasar a utilizar unas direcciones compartidas. Ej.:
 - Utilizando lectura-modificación-escritura atómicas: Intel x86, Intel Itanium, Sun Sparc
 - Utilizando LL/SC (*Load Linked / Store Conditional*): IBM Power/PowerPC

➤ Método de espera

- Método por el que un thread espera a adquirir el derecho a pasar a utilizar unas direcciones compartidas:
 - Espera ocupada (*busy-waiting*)
 - Bloqueo

➤ Método de liberación

- Método utilizado por un thread para liberar a uno (cerrojo) o varios (barrera) threads en espera

Cerrojo Simple I

- Se implementa con una variable compartida k que toma dos valores: abierto (0), cerrado (1)
- **Apertura del cerrojo**, $unlock(k)$: abre el cerrojo escribiendo un 0 (*operación invisible*)
- **Cierre del cerrojo**, $lock(k)$: Lee el cerrojo y lo cierra escribiendo un 1.
 - **Resultado de la lectura:**
 - si el cerrojo **estaba cerrado** el thread espera hasta que otro thread ejecute $unlock(k)$,
 - si **estaba abierto** adquiere el derecho a pasar a la sección crítica.
 - **leer-assignar_1-escibir en el cerrojo debe ser invisible (atómica)**

Cerrojo Simple II

- Se debe añadir lo necesario para garantizar el acceso en exclusión mutua a k y el orden imprescindible en los accesos a memoria

lock (k)

```
lock(k) {
    while (leer-asignar_1-escribir(k) == 1) {};
} /* k compartida */
```

unlock (k)

```
unlock(k) {
    k = 0 ;
} /* k compartida */
```

Cerrojos en OpenMP

Descripción	Función de la biblioteca OpenMP
Iniciar (estado unlock)	omp_init_lock (&k)
Destruir un cerrojo	omp_destroy_lock (&k)
Cerrar el cerrojo lock (k)	omp_set_lock (&k)
Abrir el cerrojo unlock (k)	omp_unset_lock (&k)
Cierre del cerrojo pero sin bloqueo (devuelve 1 si estaba cerrado y 0 si está abierto)	omp_test_lock (&k)

Cerrojos con etiqueta

- Fijan un orden FIFO en la adquisición del cerrojo (se debe añadir lo necesario para garantizar el acceso en exclusión mutua a contadores y el orden imprescindible en los accesos a memoria):

lock (contadores)

```
contador_local_adq = contadores.adq;  
contadores.adq = (contadores.adq + 1) mod max_flujos_control;  
while (contador_local_adq <> contadores.lib) {};
```

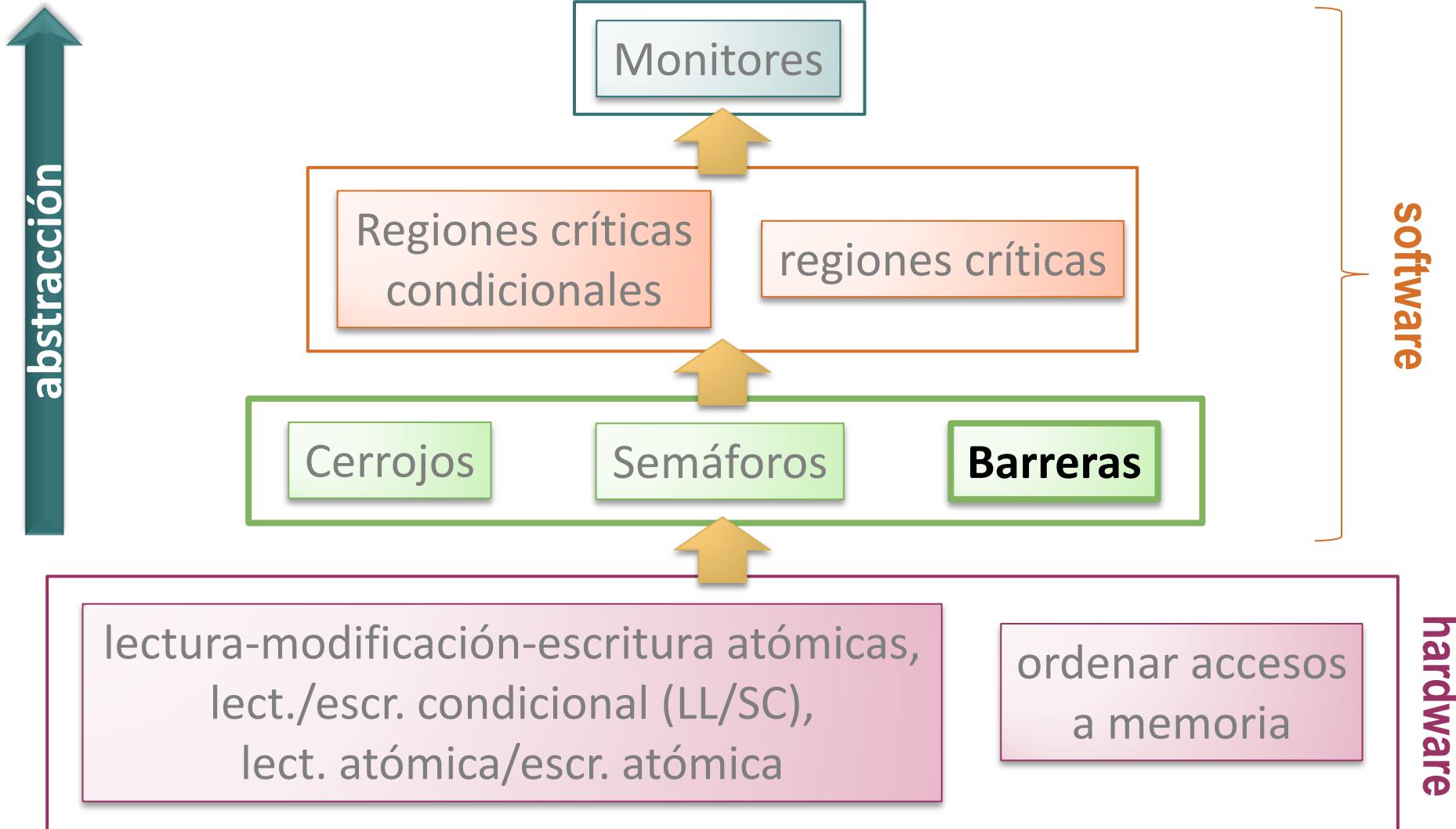
unlock (contadores)

```
contadores.lib = (contadores.lib + 1) mod max_flujos_control;
```

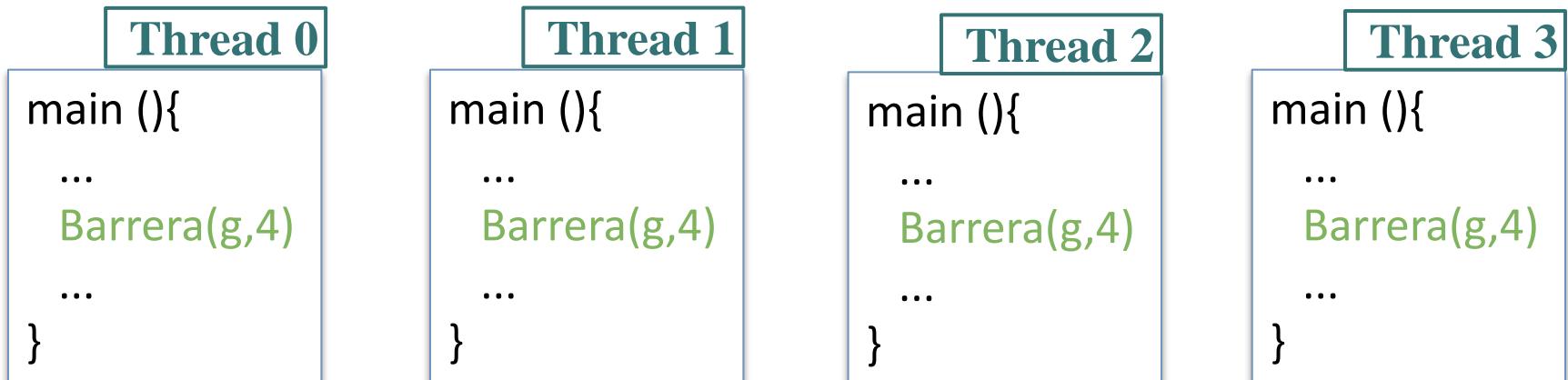
Contenido Lección 10

- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software

Soporte software y hardware de sincronización



Barreras



Barrera(id, num_threads) {

```
if (bar[id].cont==0) bar[id].bandera=0;  
cont_local = ++bar[id].cont;  
if (cont_local ==num_threads) {  
    bar[id].cont=0;  
    bar[id].bandera=1;  
}  
else espera mientras bar[id].bandera=0;  
}
```

- Acceso Ex. Mutua.

- Implementar **espera**. Si espera ocupada:

while (bar[id].bandera==0) {};

Barreras sin problema de reutilización

Barrera *sense-reversing*

```
Barrera(id, num_procesos) {
```

```
    bandera_local = !(bandera_local) //se complementa bandera local
```

```
    lock(bar[id].cerrojo);
```

```
    cont_local = ++bar[id].cont //cont_local es privada
```

```
    unlock(bar[id].cerrojo);
```

```
    if (cont_local == num_procesos) {
```

```
        bar[id].cont = 0; //se hace 0 el cont. de la barrera
```

```
        bar[id].bandera = bandera_local; //para liberar thread en espera
```

```
}
```

```
    else while (bar[id].bandera != bandera_local) {}; //espera ocupada
```

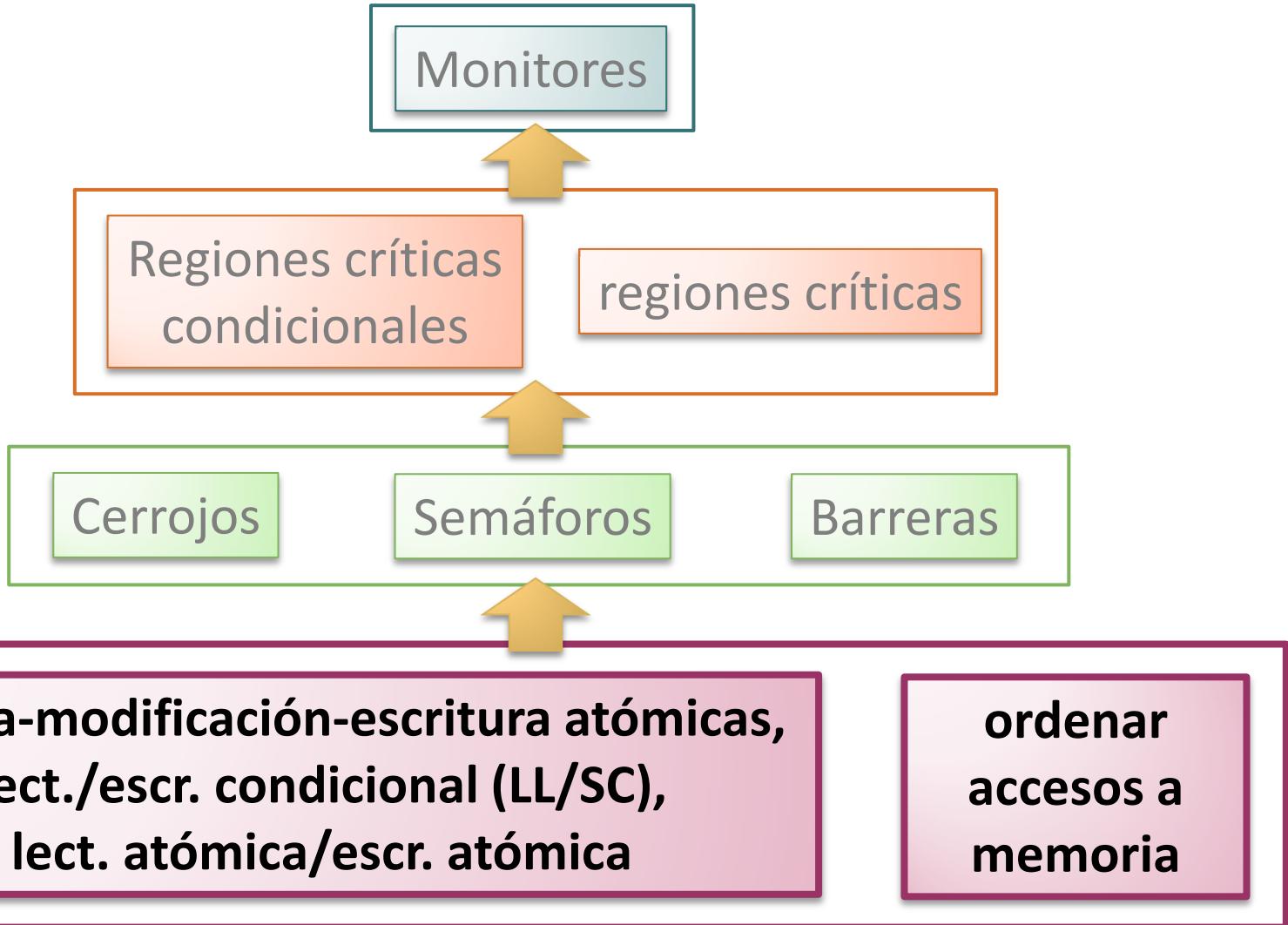
```
}
```

Contenido Lección 10

- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software
 - Instrucciones de lectura-modificación-escritura atómicas
 - Instrucciones LL/SC (*Load Linked / Store Conditional*)

Soporte software y hardware de sincronización

abstracción ↑



Instrucciones de lectura-modificación-escritura atómicas

Test&Set (x)

```
Test&Set (x) {  
    temp = x ;  
    x = 1 ;  
    return (temp) ;  
}  
/* x compartida */
```

x86

```
mov reg,1  
xchg reg,mem  
reg ↔ mem
```

Fetch&Oper(x,a)

```
Fetch&Add(x,a) {  
    temp = x ;  
    x = x + a ;  
    return (temp);  
}/* x compartida,  
a local */
```

x86

```
lock xadd reg,mem  
reg ← mem |  
mem ← reg+mem
```

Compare&Swap(a,b,x)

```
Compare&Swap(a,b,x){  
    if (a==x) {  
        temp=x ;  
        x=b; b=temp ; }  
}/* x compartida,  
a y b locales */
```

x86

```
lock cmpxchg mem,reg  
if eax=mem  
then mem ← reg  
else eax ← mem
```

Cerrojos simples con Test&Set y Fetch&Or

con Test&Set (x)

```
lock(k) {  
    while (test&set(k)==1) {};  
}  
/* k compartida */
```

x86

```
lock:    mov    eax,1  
repetir: xchg   eax,k  
          cmp    eax,1  
          jz     repetir
```

con Fetch&Oper(x,a)

```
lock(k) {  
    while (fetch&or (k,1)==1) {};  
}  
/* k compartida */
```

{ true (1, cerrado)
 false (0, abierto)

Cerrojo simple

Cerrojos simples con Compare&Swap

con **Compare&Swap(a,b,x)**

```
lock(k) {  
    b=1  
    do  
        compare&swap(0,b,k);  
    while (b==1);  
}  
/* k compartida, b local */
```

```
compare&swap(0,b,k){  
    if (0==k) { b=k | k=b; }  
}
```

Cerrojo simple en Itanium (consistencia de liberación) con Compare&Swap



lock:	//lock(M[lock])	Compare&Swap
mov ar.ccv = 0	// cmpxchg compara con ar.ccv	
	// que es un registro de propósito específico	
mov r2 = 1	// cmpxchg utilizará r2 para poner el cerrojo a 1	
spin:	// se implementa espera ocupada	
ld8 r1 = [lock] ;;	// carga el valor actual del cerrojo en r1	
cmp.eq p1,p0 = r1, r2; ;	// si r1=r2 entonces cerrojo está a 1 y se hace p1=1	
(p1) br.cond.spnt spin ;;	// si p1=1 se repite el ciclo; spnt indica que se	
	// usa una predicción estática para el salto de “no tomar”	
cmpxchg8.acq r1 = [lock], r2 ;;	//intento de adquisición escribiendo 1	
	// IF [lock]=ar.ccv THEN [lock]←r2; siempre r1←[lock]	
cmp.eq p1, p0 = r1, r2 ;;	// si r1!=r2 (r1=0) => cer. era 0 y se hace p1=0	
(p1) br.cond.spnt spin ;;	// si p1=1 se ejecuta el salto	
unlock:	//unlock(M[lock])	Consistencia de liberación
	st8.rel [lock] = r0 ;;	//liberar asignando un 0, en Itanium r0 siempre es 0

Cerrojo simple en PowerPC(consistencia débil) con LL/SC implementando Test&Set

AC ATC

lock: #lock(M[r3]) Test&Set

 li r4,1 #para cerrar el cerrojo

bucle: **lwarx** r5,0,r3 #carga y reserva: r5←M[r3]

 cmpwi r5,0 #si está cerrado (a 1)

 bne- bucle #esperar en el bucle, en caso contrario

stwcx. r4,0,r3 #poner a 1 (r4=1): M[r3] ← r4

 bne- bucle #el thread repite si ha perdido la reserva

isync #accede a datos compartidos cuando sale del bucle

Consistencia
débil

unlock: # unlock(M[r3])

sync #espera hasta que terminen los accesos anteriores

 li r1,0

stw r1,0(r3) #abre el cerrojo

Cerrojo simple en ARMv7 (consistencia débil) con LL/SC implementando Test&Set

lock:

#lock(M[r1])

Test&Set

```
    mov      r0, #1      #Para posteriormente cerrar el cerrojo asigna a r0 un 1
bucle: ldrex   r5, [r1]      #Lee cerrojo
        cmp      r5, #0      #Comprueba si el cerrojo es 0 (abierto)
        strexeq r5, r0, [r1] #Si el cerrojo está abierto intenta escribir (un 1)
        cmpeq   r5, #0      #Comprueba si ha tenido éxito la escritura
        bne     bucle       #Vuelve a intentarlo si no ha tenido éxito
        dmb          #Para asegurar que se ha adquirido el cerrojo antes de ...
                      #realizar los accesos a memoria que hay después del lock
```

Consistencia
débil

unlock:

unlock(M[r1])

```
    dmb          #Espera a que terminen los accesos anteriores ...
    mov      r0, #0      #antes de abrir el cerrojo
    str      r0, [r1]      #Abre el cerrojo
```

Cerrojo simple en ARMv8 (consistencia liberación) con LL/SC para Test&Set

lock:

```
    mov      r0, #1      #lock(M[r1])  
Test&Set  
bucle: ldaex   r5, [r1]  #Para posteriormente cerrar el cerrojo asigna a r0 un 1  
                  cmp     r5, #0  #Lee cerrojo con ordenación de adquisición  
                  strexeq r5, r0, [r1] #Si el cerrojo está abierto intenta escribir (un 1)  
                  cmpeq   r5, #0  #Comprueba si ha tenido éxito la escritura  
                  bne     bucle  #Vuelve a intentarlo si no ha tenido éxito
```

unlock:

```
    mov      r0, #0      # unlock(M[r1])  
    stl      r0, [r1]  #Abre el cerrojo usando almacenamiento con liberación  
Consistencia  
de liberación
```

Algoritmos eficientes con primitivas hardware

Suma con fetch&add

```
for (i=ithread; i<n; i=i+nthread)
    fetch&add(sum,a[i]);
/* sum variable compartida */
```

Suma con fetch&add

```
for (i=ithread; i<n; i=i+nthread)
    sump = sump + a[i];
fetch&add(sum,sump);
/* sum variable compartida */
```



Suma con compare&swap

```
for (i=ithread; i<n; i=i+nthread)
    sump = sump + a[i];
do
    a = sum;
    b = a + sump;
    compare&swap(a,b,sum);
    while (a!=b);
/* sum variable compartida */
```

Para ampliar ...

➤ Webs

- Implementación en el kernel de linux de cerrojos con etiqueta <http://lxr.free-electrons.com/source/arch/x86/include/asm/spinlock.h>

➤ Artículos en revistas

- Graunke, G.; Thakkar, S.; , "Synchronization algorithms for shared-memory multiprocessors," *Computer* , vol.23, no.6, pp.60-69, Jun 1990. Disponible en (biblioteca ugr):
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=55501&isnumber=2005>