

Ejercicio 5. Supongamos que se va a ejecutar en paralelo el siguiente código (initialmente x e y son 0):

P1	P2
x=30;	while (flag==0) {};
y=40;	r1=x;
flag=1; Siuc	r2=y;

Siempre lo sostenemos

Qué datos puede obtener P2 en los registros r1 y r2 si (considere que el compilador no altera el código):

- (a) Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.
- (b) Se ejecutan en un multiprocesador con un modelo de consistencia que relaja todos los órdenes en los accesos a memoria. Razoné su respuesta.

a) $x=0, y=0$

$$r_1 = 30$$

$$r_2 = 40$$

W_x
W_y
W_f

P1
x=30;
y=40;
flag=1; Siuc

P2
while (flag==0) {};
r1=x; R _x
r2=y; R _y

Jordán el resultado

S. en P2 flag=1 entonces W_x y W_y se han hecho luego
tenemos garantizado el siguiente orden
 $W_x(60), W_y(40), W_f(1), R_j(1), R_j(30), R_j(40)$

b) $x=0, y=0$

Ejecución: $r_1 =$
1 (secuenc) 30

$r_2 =$
40

W_x
W_y
W_f

P1
x=30;
y=40;
flag=1; Siuc

P2
while (flag==0) {};
r1=x; R _x
r2=y; R _y

R_f
R_x
R_y

2 (nogar orden) 30 0

dado f=1 antes de que y=40 luego el orden de lectura
sería

$$\left. \begin{array}{l} R_y(0) W_y(40) \\ R_f(1), R_x(30) \end{array} \right\} \cdot W_y(40) R_y(40)$$

3 (nogar orden) 30 40

dado f=1 antes de que x=30, y=40 luego el orden de lectura

sería

$$\left. \begin{array}{l} R_x(0) R_y(0) \\ R_x(0) R_y(40) (W_y adelante W_x) \\ R_x(30) R_y(0) (W_x antes W_y) \end{array} \right\}$$

4 (nogar orden) 30 0

dado f=1 antes de que x=30, y=40 luego el orden de lectura

sostenemos que el orden de escritura sea la que hay
pero que R_y adelante R_x. dado

$$\left. \begin{array}{l} \text{diseño correcto} \\ R_f(1) \end{array} \right\} \cdot R_x(30) R_y(0)$$

Ejercicio 4. Supongamos que se va a ejecutar en paralelo el siguiente código, donde x e y son variables compartidas (initialmente x e y son 0):

P1	P2
$x=1;$	$y=1;$
$x=2;$	$y=2;$
print y ;	print x ;

Qué resultados se pueden imprimir si (considere que el compilador no altera el código):

- (a) Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.
- (b) Se ejecutan en un multiprocesador basado en un bus que garantiza todos los órdenes excepto el orden W→R. Esto es debido a que los procesadores tienen buffer de escritura, permitiendo el procesador que las lecturas en el código que ejecuta adelantan a las escrituras que tiene su buffer. Obsérvese que hay varios resultados posibles.

W_x	P1	P2	W_y
W_x	$x=1;$	$y=1;$	
W_x	$x=2;$	$y=2;$	
R_y	print y ;	print x ;	R_x

a) $P_1(y)$

$P_2(x)$

2

0

2

1

2

2

0

2

1

2

2

2

Supongamos P_1 impriera primero luego los posibles

valores son

$x=0$

$x=1$

$x=2$

Si $x=0 \Rightarrow$ se ha ejecutado P_1

Si $x=1 \Rightarrow$ se ha ejecutado la primera linea de P_1

Si $x=2 \Rightarrow$ se ha ejecutado la segunda linea de P_1

Analogamente con y

b) Pueden darse las siguientes combinaciones con:

$P_1(y)$

$P_2(x)$

0

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

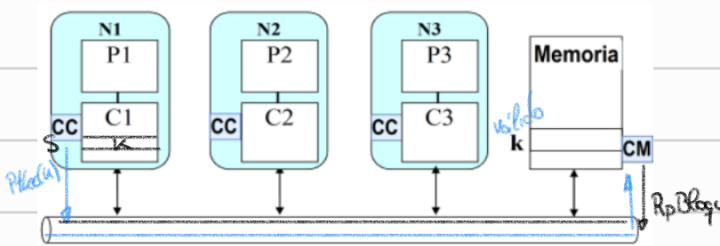
1

1

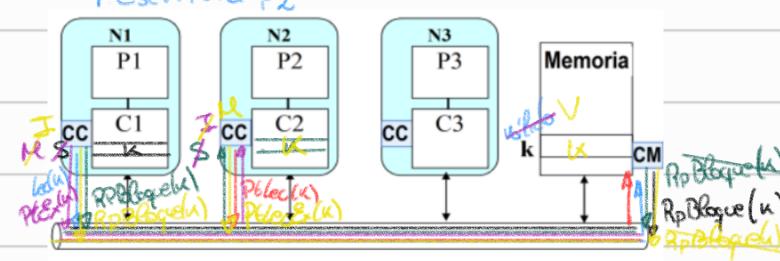
Ejercicio 1. En un multiprocesador SMP con 4 procesadores o nodos (N0-N3) basado en un bus, que implementa el protocolo MSI para mantener la coherencia, supongamos una dirección de memoria incluida en un bloque que no se encuentra en ninguna cache. Indique los estados de este bloque en las caches y las acciones que se producen en el sistema ante la siguiente secuencia de eventos para dicha dirección:

1. Lectura generada por el procesador 1
2. Lectura generada por el procesador 2
3. Escritura generada por el procesador 1
4. Escritura generada por el procesador 2
5. Escritura generada por el procesador 3

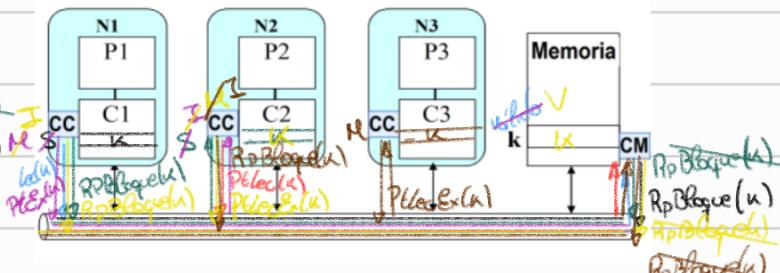
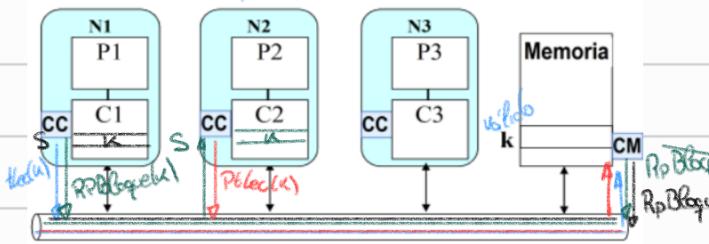
1. Lectura por P1 → P1Loc₁(u) → P1Loc₂(u)



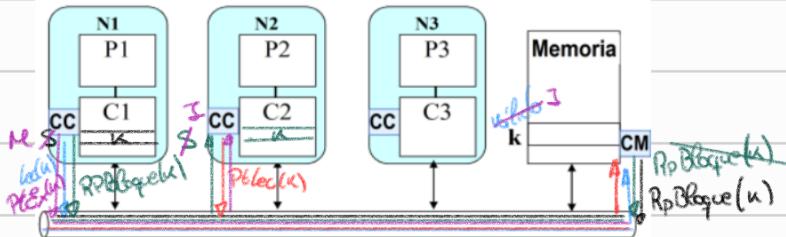
4. Escritura P2(u) → P1Loc₂(u) → P1Loc₃(u)



2. Lectura por P2 → P1Loc₂(u) → P1Loc₃(u)



3. Escritura P1 → P1Loc₁(u) → P1Loc₂(u)



Ejercicio 2. Para un multiprocesador de memoria distribuida con 8 nodos se quiere implementar un protocolo para mantenimiento de coherencia basado en directorios. Suponiendo que se necesitan un bit de estado para un bloque en el directorio de memoria principal y que el tamaño de una línea de cache es de 64 bytes, calcular el porcentaje del tamaño de memoria principal que supone el tamaño del directorio de vector de bits completo.

$$Nº\ filas = \frac{Nº\ bloques}{Tamaño\ linea} = \frac{Tamaño\ linea}{Tamaño\ linea}$$

$$Nº\ columnas = 1\ bit + 8\ bits = 9\ bits$$

9
3 por cada

$$\frac{Tamaño\ linea}{Tamaño\ memoria} \times 100 = \frac{\frac{Tamaño\ linea}{Tamaño\ linea} \cdot Tamaño\ linea}{Tamaño\ memoria} \cdot 100 = \frac{Tamaño\ linea}{Tamaño\ memoria} \cdot 100$$

$$= 176\%$$

$$100 = \frac{9}{64} \cdot 100$$

Ejercicio 3. Suponga que en un CC-NUMA de red estática de 4 nodos (N0-N3) se implementa un protocolo MSI basado en directorios sin difusión con dos estados en el directorio (válido e inválido). Cada nodo tiene 8 GBytes de memoria y una línea de cache supone 64 Bytes. Considere que el directorio utiliza vector de bits completo. (a) Calcule el tamaño del directorio de uno nodo en bytes. (b) Indique cual sería el contenido del directorio, las transiciones de estados (en cache y en el directorio) y la secuencia de paquetes generados por el protocolo de coherencia en los siguientes accesos sobre una dirección D que se encuentra en la memoria del nodo 3 (initialmente D no está en ninguna cache):

1. Lectura generada por el procesador del nodo 1
2. Escritura generada por el procesador del nodo 1
3. Lectura generada por el procesador del nodo 2
4. Lectura generada por el procesador del nodo 3
5. Escritura generada por el procesador del nodo 0

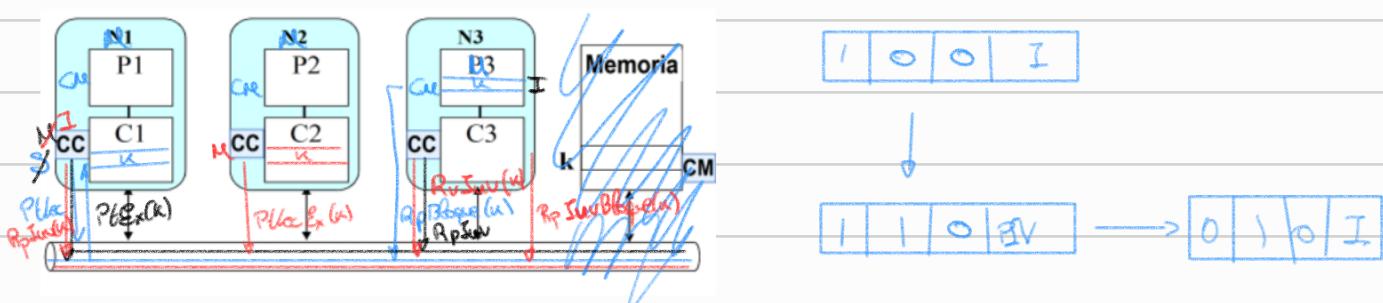
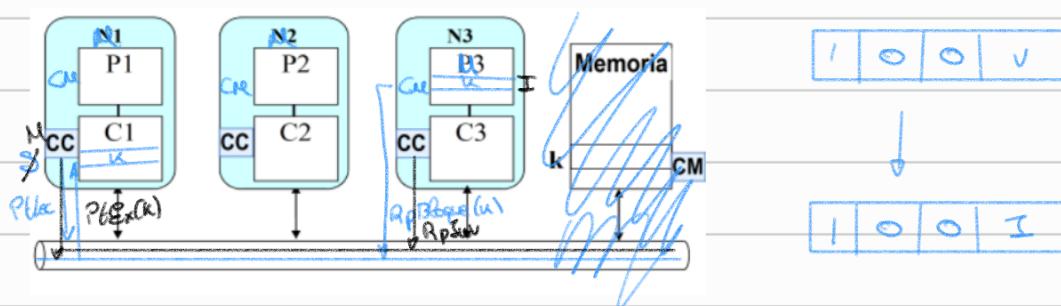
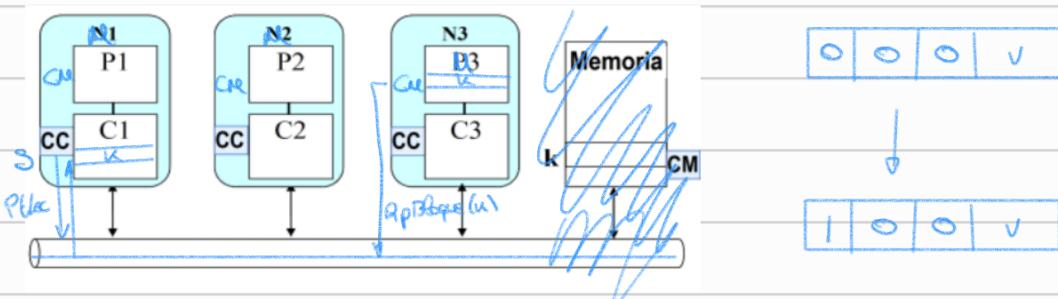
$$a) N_{\text{filas}} = \frac{T_{\text{cache}}}{T_{\text{dirección}}} = \frac{2^{23}}{2^6} = 2^{17}$$

$$T_{\text{dirección}} = N_{\text{filas}} \cdot T_{\text{drl}} = 2^{17} \cdot 5\text{B} = 2^{24} \cdot 5\text{B}$$

$$T_{\text{drl}} = n^{\circ} \text{ nodos} + n^{\circ} \text{ bits estado} = 4 + 1 = 5\text{B}$$

b)

1. P1 Lec_x(x)



Ejercicio 6. Se quiere implementar un cerrojo simple en un multiprocesador SMP basado en procesadores de la línea x86 de Intel, en particular, procesadores Intel Core. (a) Teniendo en cuenta el modelo de consistencia de memoria que ofrece el hardware de este multiprocesador ¿podríamos implementar la función de liberación del cerrojo simple usando "mov k, 0", siendo k la variable cerrojo? Razona su respuesta. (b) ¿Cómo se debería implementar la función de liberación de un cerrojo simple si se usan procesadores con la arquitectura ARMv7? Razona su respuesta.

Multithread simultáneo

x86 . W → R

a) Un cerrojo simple se realiza de la siguiente forma

void locu(u);

void uelocu(u);

while (u==1) {; if (u==1)

u=0

{

{

Donde disponemos de la siguiente secuencia de accesos:

locu. R_k, W_k(1)

uban. W_k(0)

Como en x86 sólo se refleja el orden W→R y no hay ninguna lectura posterior a escritura el corcho funcionará a la perfección.

En DSM la sentencia está expresada en código C en la función uban de la siguiente manera:

"n=0;"

b) Como en este caso DSM relaja todos los órdenes, para no permitir que haya adelantamientos pues esto provocaría errores de ejecución la implementación querría de la siguiente forma.

void locu(u){}

while (u==1) {;f

climb

u=1;

{

void uban(u){

u=0;

{

Ejercicio 7. Se ha ejecutado el siguiente código en un multiprocesador con un modelo de consistencia que no garantiza ni W→R ni W→W (garantiza el resto de órdenes):

```
(1) sump = 0;  
(2) for (i=ithread ; i<8 ; i=i+nthread) {  
(3)     sump = sump + a[i];  
}  
(4) while (Fetch_&_Or(k,1)==1) {};  
(5) sum = sum + sump;  
(6) k=0;
```

Conteste a las siguientes preguntas (considere que el compilador no altera el código):

- (a) Indique qué se puede obtener en sum si se suma la lista a=[1,2,3,4,5,6,7,8]. k y sum son variables compartidas que están inicialmente a 0 (el resto de variables son privadas), nthread = 3, ithread es el identificador del thread en el grupo (0,1,2). Si hay varios posibles resultados, se tienen que dar todos ellos. Justifique su respuesta.
- (b) ¿Qué resultados se pueden obtener si lo único que no garantiza el modelo de consistencia es el orden W→R? Justifique su respuesta.

a)	1	2	3	4	5	6	7	8
----	---	---	---	---	---	---	---	---

k=0

sum=0

uthread=3

Corcho

W_k(0) → (1) sump = 0;

(2) for (i=ithread ; i<8 ; i=i+nthread) {
 sump = sump + a[i];
}

(4) while (Fetch_&_Or(k,1)==1) {};

(5) sum = sum + sump;

(6) k=0;

R_k, R_{sum}, W_k

R_k, R_{sum}, W_{sum}

R_k, W_k(1)

R_k, R_{sum}, W_{sum}

W_k(0)

A priori sabemos que las líneas 2 y 3 son inmutables y vanjuntas pues son escrituras condicionales. También sabemos que la linea 4 es atómica.

En este caso hay varias soluciones. Supongamos que no hay sobreescrituras:

it 0 1 2 3 4 5 6 7

lebra 0 1 2 0 1 2 0 1

Este es el reparto realizado. Descomponemos por l_i a la lebra con identificador, por var_i a la variable privada var de la lebra l_i . Por último descomponemos por f_{l_i} el acto de que l_i entre en una región de código que implementa un corajo

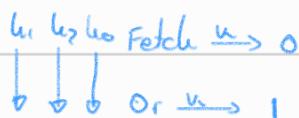
Veamos este ejemplo

$$\text{sump}_{l_0} = 0 \rightarrow \text{sump}_{l_0} + 1 = 1 \rightarrow \text{sump}_{l_0} + 4 = 5 \rightarrow \text{sump}_{l_0} + 7 = 12$$

$$\text{sump}_{l_1} = 0 \rightarrow \text{sump}_{l_1} + 2 = 2 \rightarrow \text{sump}_{l_1} + 5 = 7 \rightarrow \text{sump}_{l_1} + 8 = 15$$

$$\text{sump}_{l_2} = 0 \rightarrow \text{sump}_{l_2} + 3 = 3 \rightarrow \text{sump}_{l_2} + 6 = 9$$

$k = 0$



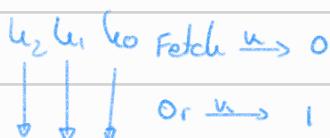
$$\text{sum} = \text{sum} + \text{sump}_{l_1} = 15 \rightarrow \text{sum} = \text{sum} + \text{sump}_{l_2} = 27 \rightarrow \text{sum} = \text{sum} + \text{sump}_{l_0} = 36$$

$k = 0$

Dando así el resultado correcto

Supongamos ahora que la escritura de la línea (6) sobrepasa a la de la línea (5) dentro de la
el corajo

$k = 0$



$$\text{sum} = 0$$

$$\text{sump}_{l_2} = 9$$

$k = 0$ (entre l_1)

$$\text{sum} = 0$$

$$\xrightarrow{\text{escribel}_1} \text{sump}_{l_1} = 17$$

$$\text{sum} = 17$$

$k = 0$ (entre l_0)

$$\xrightarrow{\text{escribel}_2} \text{sum} = 9$$

$$\sum_{i=1}^n p_i = 1$$

$$\sum_{i=1}^n = 21$$

Después jugando con ambas hebras da lugar a una gran cantidad de posibilidades causadas por el adibutamiento de una escritura a otra.

b) Si no se garantiza \rightarrow todo funcionaría a la perfección

Ejercicio 8. ¿Qué ocurre si en el segundo código para implementar barreras visto en clase eliminamos la variable local, `cont_local`, sustituyéndola en los puntos del código donde aparece por el contador compartido asociado a la barrera `bar[id].cont`?

```
1 void barrera(int id, int num_threads){  
2     // Variable privada de cada hebra,  
3     // complementa la bandera anterior  
4     int bandera_local = !bandera_local;  
5  
6     // Acceso a sección crítica  
7     lock(barreras[id].cerrojo);  
8     // cuenta hebra hebra  
9     // Aumenta el contador y guarda su valor en variable privada  
10    int cuenta_local = ++barreras[id].contador;  
11    barreras[id].bandera = bandera_local;  
12    unlock(barreras[id].cerrojo);  
13    // Si todas las hebras han llegado  
14    if(cuenta_local == num_threads){  
15        barreras[id].contador = 0; // Para un próximo uso  
16  
17        // Si todas han llegado, todas tienen la misma bandera local  
18        barreras[id].bandera = bandera_local;  
19    } else{  
20        // Pone a la hebra en espera  
21        while(barreras[id].bandera != bandera_local) {}  
22    }  
23 }
```

struct Barrera {

int counter $\oplus 1 \oplus 3$

int flag 0

Repite loc

adq $\oplus 1 \oplus 3$

lib $\oplus 1 \oplus 3$

Haremos para ello alguna pequeña ejecución.

Supongamos que hay tres hebras ejecutando el código de la barra.

h ₁	h ₂	h ₃	num_threads=3
0	0	0	
1	1	1	

Por lo que no hay ningún error, es decir, se ejecuta bien. En clase se comentó que debía ser privada; no recuerdo el por qué pero era así!

Ejercicio 9. Suponiendo que la arquitectura dispone de instrucciones Fetch&Add, simplifique el segundo código para barreras visto en clase.

Lo escribiré al completo haciendo uso de las siguientes estructuras

struct Lock {

struct Barrier {

int adq, lib,

int counter, flag,

{

Lock loc;

{

```

void barriera (int idt, int num_threads) {
    int bandera_local = 1 bandera_local
    while ((Fetch & Add (barrieras[idt] counter, 1)) == num_threads) {
        barrieras[idt] counter = 0;
        barrieras[idt] flag = bandera_local;
    }
}

```

Ejercicio 10. Se quiere parallelizar el siguiente ciclo de forma que la asignación de iteraciones a los procesadores disponibles se realice en tiempo de ejecución (dinámicamente):

```

For (i=0; i<100; i++) {
    Código que usa i
}

```

Nota: Considerar que las iteraciones del ciclo son independientes, que el único orden no garantizado por el sistema de memoria es W->R, que las primitivas atómicas garantizan que sus accesos a memoria se realizan antes que los accesos posteriores y que el compilador no altera el código.

- (a) Parallelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva Fetch&Or para garantizar exclusión mutua.
- (b) Parallelizar el anterior ciclo en un multiprocesador que además tiene la primitiva Fetch&Add.

a) while (Fetch & Or (u, 1) == 1) {

u=u ; i=i+1;

u=0

while (u<100) {

código para u(i)

while (Fetch & Or (u, 1) == 1) {, };

u=i ; i=i+1;

u=0

{

b) u=Fetch & Add (i, 1);

while (u<100) {

código para u(i);

u=Fetch & Add (i, 1);

{

Ejercicio 11. Un programador está usando el siguiente código para barreras (bar es un vector compartido, k es una variable compartida, el resto son variables locales, Fetch_&_Or(k, 1) realiza sus accesos a memoria antes de que puedan realizarse los accesos posteriores):

```

Barrera(id, num_procesos)
{
    band_local = !(band_local)
    while (Fetch_&_Or(k, 1) == 1) {};
    cont_local = ++bar[id].cont;
    R_b1 W_b1 atomico
    R_b2 W_b2
    R_c1 W_c1
    R_c2 W_c2
    R_u1 W_u1
    R_u2 W_u2
    if (cont_local == num_procesos)
        bar[id].cont = 0;
        bar[id].band = band_local;
    else while (bar[id].band != band_local) {};
}

```

Conteste a las siguientes cuestiones (considere que el compilador no altera el código):

(a) ¿Funciona bien este código como barrera en un multiprocesador en el que lo único que no garantiza su modelo de consistencia es el orden W->R? Razone por qué.

(b) Funciona bien este código como barrera en un multiprocesador con modelo de consistencia que no garantice ningún orden en los accesos a memoria? Razone por qué.

a) Nos piden analizar el comportamiento en un procesador similar a un x86

Sí: no hay conflictos escritura escritura pero sí lectura escritura.

Sí: la lectura de la línea 6 adelante a la escritura de la línea 4 no ocurriría nunca puesto

que ninguna falta de consistencia; el fallo escritura escritura adelantado a la escritura de cont_local de la línea 3 pues cont_local no toucharía el dato actualizado. No obstante, al ser

de la misma posición de memoria se hace el valor que se vaya a escribir. Por tanto, no habrá fallo.

Tampoco causalía fallo que la lectura de bandera_local de la línea (7) adelante a la escritura de la línea (6) pues no son de la misma variable y no hay situaciones comprometedoras.

b) En este caso sí hay fallo y no funcionaría pues simplemente que la escribe sobre cont_local adelante a la escritura de bar[id].cont occasionaría una situación de inconsistencia indecisa pues el contador local seguirá en el mismo valor provocando una espera ocupada de todos los hilos.

Ejercicio 12. Se quiere implementar un programa que calcule en paralelo la siguiente expresión en un multiprocesador en el que sólo se relaja el orden W>R y en el que sólo se dispone de primitiva de sincronización test_&_set:

$$d = \frac{1}{N} \sum_{i=1}^N x_i^2 - \bar{x}^2, \text{ donde } \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

Un programador ha implementado el código de abajo. Tenga en cuenta lo siguiente: el código lo ejecutan nthread threads en paralelo; ithread es una variable local que nota el identificador del thread; i, med y vari son variables locales; i, med, vari, el vector x y N son variables compartidas; inicialmente med, vari, medl y vari l son 0.

```
(1) for (i=ithread; i<N; i=i+nthread) {  
(2)     medl=medl+x[i];  
(3)     vari=vari+x[i]*x[i];  
(4) }  
(5) med = med + medl/N; vari = vari + vari/N;  
(6) vari= vari - med*med;  
(7) if (ithread==0) printf("varianza = %f", vari); //imprime en pantalla
```

Conteste a las siguientes cuestiones (considere que el compilador no altera el código):

- (a) Se ha ejecutado este código usando varios threads y se ha visto que, aunque N y el vector x no varían, no siempre se imprime lo mismo. ¿Por qué ocurre esto?

(b) Añada lo mínimo necesario para solucionar el problema teniendo en cuenta que sólo se dispone para implementar sincronización de test_&_set (tampoco se dispone de primitivas software de sincronización). Indique qué variables son ahora compartidas y cuáles locales.

(c) Escriba el programa suponiendo que el multiprocesador además tiene primitivas de sincronización fetch_&_add (se tendrá en cuenta las prestaciones). Indique qué variables son compartidas y cuáles locales.

(d) Escriba el programa ahora suponiendo que el multiprocesador sólo tiene primitivas de sincronización compare_&_swap (se tendrá en cuenta las prestaciones). Indique qué variables son compartidas y cuáles locales.

- Hay dos tipos de errores:
 1. No se accede en exclusión mutua a las variables compartidas med y vari por parte de los diferentes threads(5). Esto ocasiona que puedan intentar varios threads a la vez acumular el resultado parcial que han calculado provocando una condición de carrera bastante notoria.
 2. Las operaciones de la línea (6) leen y modifican la variable vari. Tal y como está el código, esa operación la realizan todos los threads lo que puede llevar a restar varias veces a vari el resultado de elevar al cuadrado med. Para evitar este problema se puede escribir en vari o meter (6) dentro del if que acompaña a printf.
 3. El thread 0 obtiene el valor definitivo a partir de med y vari e imprime sin esperar que todos los threads acumulen en las variables locales medl y vari los resultados parciales que han obtenido en el bucle en las variables medl y vari. Esto supone que cuando imprime pueden haber intentado acumular su resultado parcial el thread 0 y una combinación del resto de threads en un numero de 0 a nthreads-1. Por este motivo, aunque se accediera en exclusión mutua a las variables compartidas, se podrían imprimir distintos resultados en distintas ejecuciones.
- Se accederá en exclusión mutua a med y vari implementando un cerrojo simple con una variable compartida k1, se tiene que añadir una barrera antes de que imprima el thread 0 y se introduce (6) dentro del if. La barrera se debe implementar también usando un cerrojo simple:

```
for (i=ithread, j=N; i+j=nthread){  
    medl=medl+x[i];  
    vari=vari+x[i]*x[i];  
}  
medl=medl/n; vari=vari/N;  
while((test&set(k1))){ //lock(k1)  
    med = med+medl; vari=vari+vari;  
    k1=0;  
    bandera_local=!bandera_local;  
    while (test&set(k2)){  
        bar[id].cont+=1;cont_local=bar[id].cont;  
        k2=0;  
        if(cont_local == num_procesos){  
            bar[id].cont = 0;  
            bar[id].flag = bandera_local;  
        }  
        else while (bar[id].flag!=bandera_local){  
            if(iThread==0){vari=vari-med*med; printf...}
```

Los ámbitos de las variables son los ámbitos lógicos que uno puede pensar.

- Con fetch&add(), para el acceso en exclusión mutua no es necesario usar variables compartidas extra como cerrojos:

```
for (i=ithread, i<N; i+=nthread){  
    medi=medi+x[i];  
    vari=vari+x[i]*x[i];  
}  
fatch&add(med,medi/N);fetch&add(vari,vari/N);  
bandera_local=!bandera_local;  
cont_local=feccth&add(bar[id].cont,1);  
if(cont_local == num_procesos){  
    bar[id].cont = 0;  
    bar[id].flag = badera_local;  
}  
else while (bar[id].flag!=bandera_local){  
    if(ithread==0){ vari=vari-med*med; printf...}  
}
```
- Con compare&swap(), para el acceso en exclusión mutua no es necesario usar variables compartidas extra como cerrojos:

```
for (i=ithread, i<N; i+=nthread){  
    medi=medi+x[i];  
    vari=vari+x[i]*x[i];  
}  
medi=medi/N; vari=vari/N;  
do{  
    a=med;  
    b=a+medi;  
    compare&swap(a,b,medi);  
}while(a!=b);  
do{  
    a = vari;  
    b= a+vari;  
    compare&swap(a,b,vari);  
}while(a!=b);  
  
bandera_local=!bandera_local;  
do{  
    cont_local = bar[id].cont;  
    b=cont_local+1;  
    compare&swap(cont_local,b,bar[id].cont);  
}while(cont_local!=b);  
if(cont_local == num_procesos-1){  
    bar[id].cont = 0;  
    bar[id].flag = badera_local;  
}  
else while (bar[id].flag!=bandera_local){  
    if(ithread==0){ vari=vari-med*med; printf...}  
}
```