2º curso / 2º cuatr.

Grado en Ing. Informática

# Arquitectura de Computadores. Ejercicios resueltos

## Tema 3. Arquitecturas con paralelismo a nivel de thread (TLP)

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

## 1 Ejercicios

**Ejercicio 1.** En un multiprocesador SMP con 4 procesadores o nodos (N0-N3) basado en un bus, que implementa el protocolo MESI 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

### Solución

Datos del ejercicio

Se accede a una dirección de memoria cuyo bloque k no se encuentra en ninguna cache, luego debe estar actualizado en memoria principal y el estado en las caches se considera inválido.

Estado del bloque en las **caches** y acciones generadas ante los eventos que se refiere a dicho bloque

Hay 4 nodos con cache y procesador (N0-N3). Intervienen N1, N2 y N3. En la tabla se van a utilizar las siguentes siglas y acrónimos:

MP: Memoria Principal.

PtLec(k): paquete de petición de lectura del bloque k.

PtLecEx(k): paquete de petición de lectura del bloque k y de petición de acceso exclusivo al bloque k.

RpBloque(k): paquete de respuesta con el bloque k.

Se va a suponer que no existe en el sistema paquete de petición de acceso exclusivo a un bloque sin lectura (no existe PtEx).

| ESTADO INICIAL | EVENTO | ACCIÓN | ESTADO SIGUIENTE |
|---|---|---|---|
| **N1) Inválido**<br>**N2) Inválido**<br>**N3) Inválido** | P1 lee k | 1.- N1 (el controlador de cache de N1) genera y deposita en el bus una petición de lectura del bloque k (PtLec(k)) porque no lo tiene en su caché válido<br>2.-MP (el controlador de memoria de MP), al observar PtLec(k) en el bus, genera la respuesta con el bloque (RpBloque(k)). | **N1) Exclusivo**<br>**N2) Inválido**<br>**N3) Inválido** |

| | | | |
|---|---|---|---|
| | | 3.- N1 (el controlador de cache de N1) recoge del bus la respuesta depositada por la memoria principal (RpBloque(k)), el bloque entra en la cache de N1 en estado exclusivo ya que no hay copia en otra cache del bloque (es decir, la salida de la OR cableada con entradas procedentes de todas las caches es 0). | |
| **N1) Exclusivo**<br>**N2) Inválido**<br>**N3) Inválido** | P2 lee k | 1.- N2 genera y deposita en el bus una PtLec(k) porque no tiene k en su caché en estado válido<br>2.- N1 observa PtLec(k) en el bus y, como tiene el bloque en estado exclusivo, lo pasa a compartido (la copia que tiene ya no es la única válida en caches). MP, al observar PtLec(k) en el bus, genera la respuesta con el bloque (RpBloque(k)).<br>3.- N2 recoge RpBloque(k) que ha depositado la memoria, el bloque entra en estado compartido en la cache de N2 (la salida de la OR cableada será 1). | **N1) Compartido**<br>**N2) Compartido**<br>**N3) Inválido** |
| **N1) Compartido**<br>**N2) Compartido**<br>**N3) Inválido** | P1 escribe en k | 1.- N1 genera petición de lectura con acceso exclusivo del bloque k (PtLecEx(k)) (suponemos que no hay petición de acceso exclusivo sin lectura, no hay PtEx). N1 modifica la copia de k que tiene en su cache y lo pasa a estado modificado.<br>2.- N2 observa PtLecEx(k) y, como la petición incluye acceso exclusivo (Ex) a un bloque que tiene en su cache en estado compartido, pasa su copia a estado inválido. MP genera RpBloque(k) porque observa en el bus una petición de k con lectura (Lec), pero esta respuesta no se va a recoger del bus. N1 no recoge RpBloque(k) depositada por la memoria porque tiene el bloque válido. | **N1) Modificado**<br>**N2) Inválido**<br>**N3) Inválido** |
| **N1) Modificado**<br>**N2) Inválido**<br>**N3) Inválido** | P2 escribe en k | 1.- N2 genera petición de lectura con acceso exclusivo de k (PtLecEx(k))<br>2.- N1 observa PtLecEx(k) y, como tiene el bloque en estado modificado (es la única copia válida en todo el sistema), inhibe la respuesta de MP y genera respuesta con el bloque RpBloque (k), y además, como el paquete pide acceso exclusivo a k (Ex), invalida su copia del bloque k.<br>3.- N2 recoge RpBloque(k), introduce k en su cache, lo modifica y lo pone en estado modificado | **N1) Inválido**<br>**N2) Modificado**<br>**N3) Inválido** |
| **N1) Inválido**<br>**N2) Modificado**<br>**N3) Inválido** | P3 escribe en k | 1.- N3 genera petición de lectura con acceso exclusivo de k PtLecEx(k)<br>2.- N2 observa PtLecEx(k) y, como tiene el bloque en estado modificado, inhibe la respuesta de MP y genera respuesta con el bloque RpBloque (k), y además, como el paquete pide acceso exclusivo a k (Ex), invalida su copia de k.<br>3.- N3 recoge RpBloque(k), introduce el k en su cache, lo modifica y lo pone en estado modificado | **N1) Inválido**<br>**N2) Inválido**<br>**N3) Modificado** |

◀

**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.

**Solución**

Datos del ejercicio:

- Tamaño Línea de Cache (bloque de memoria): 64 bytes = TLC
- 1 bit de estado por bloque en el directorio

Directorio de vector de bits completo

Tamaño_por_nodo = Nº_de_bloques_memoria_nodo × Espacio_por_bloque

TMN: Tamaño Memoria principal por Nodo

Teniendo en cuenta que hay 8 nodos y un bit de estado, se necesitan 9 bits por entrada del directorio.

$$Tamaño\_directorio\_por\_nodo = \frac{TMN}{TLC} \times (8b + 1b)$$

$$\%\_de\_TMN = \frac{\frac{TMN}{TLC} \times 9b}{TMN} \times 100 = \frac{9b}{TLC} \times 100 = \frac{9b}{64 \times 8b} \times 100 \approx 1{,}76\%$$

**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 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 (inicialmente 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

**Solución**

Datos del ejercicio

- Tamaño Memoria principal por Nodo: 8GB = TMN
- Tamaño Línea de Cache (bloque de memoria): 64 B = TLC
- 1 bits de estado por bloque en el directorio ya que hay que codificar dos estados (válido, inválido).
- Se accede a una dirección de la memoria del nodo 3 cuyo bloque no se encuentra en ninguna cache, luego debe estar actualizado en memoria principal.

**(a)**

$$Tamaño\_por\_nodo = \frac{TMN}{TLC} \times (1b + 4b) = \frac{2^{33}B}{2^6 B} \times 5b = 2^{27} \times 5b = 2^{20} \times \frac{2^7 \times 5b}{2^3 b / B} = (2^{20} \times 2^4 \times 5)B = 80MB$$

$$Tamaño\_por\_nodo = \frac{TMN}{TLC} \times (2b + 4b) = \frac{2^{33}B}{2^6 B} \times 6b = 2^{27} \times 6b = 2^{20} \times \frac{2^7 \times 5b}{2^3 b / B} = (2^{20} \times 2^4 \times 6)B = 96MB$$

$$Tamaño_{directorio} = Tamaño_{por_{nodo}} \times 4 = 384MB$$

**(b)**

Intervienen los nodos N0, N1, N2 y N3. V es Válido e I inválido. BD denota el bloque de la dirección D.

| ESTADO INICIAL | EVENTO | ACCIÓN | ESTADO SIGUIENTE |
|---|---|---|---|
| **N0) Inválido**<br>**N1) Inválido**<br>**N2) Inválido**<br>**N3) Inválido**<br>**D) Válido**<br>V \| \| \| \| \| | P1 lee D | 1. N1 envía petición de lectura del bloque BD (PtLec(BD)) a N3<br>2. N3 recibe PtLec(BD). Como tiene el bloque BD en estado Válido, (1) envía paquete de respuesta con el bloque a N1 (RpBloque(BD)) y (2) pone el bit de N1 en el directorio a 1.<br>3. N1 recibe RpBloque(BD) y pone el bloque en cache en estado Compartido | **N0) Inválido**<br>**N1) Compartido**<br>**N2) Inválido**<br>**N3) Inválido**<br>**D) Válido**<br>V \| \| 1 \| \| \| |
| **N0) Inválido**<br>**N1) Compartido**<br>**N2) Inválido**<br>**N3) Inválido**<br>**D) Válido**<br>V \| \| 1 \| \| \| | P1 escribe en D | 1. N1 envía petición de acceso exclusivo para BD (PtEx(BD)) a N3.<br>2. N3, cuando recibe PtEx(BD), (1) pasa BD a estado Inválido y (2) envía paquete de respuesta a N1 confirmando invalidación (RpInv(BD)).<br>3. N1, recibida la respuesta, modifica el bloque y lo pasa a estado Modificado. | **N0) Inválido**<br>**N1) Modificado**<br>**N2) Inválido**<br>**N3) Inválido**<br>**D) Inválido**<br>I \| \| 1 \| \| \| |
| **N0) Inválido**<br>**N1) Modificado**<br>**N2) Inválido**<br>**N3) Inválido**<br>**D) Inválido**<br>I \| \| 1 \| \| \| | P2 lee D | 1. N2 envía PtLec(BD) a N3 porque no tiene BD.<br>2. N3, como tiene BD en estado Inválido: (1) reenvía (RvLec(BD)) la petición al nodo N1 (que según el directorio tiene copia válida del bloque), y (2) pone en la entrada del bloque en el directorio estado pendiente de Válido y activa el bit de N2.<br>3. N1 recibe RvLec(BD) y: (1) envía a N3 un paquete de respuesta con el bloque (RpBloque(BD)), y (2) pasa BD en su cache a Compartido.<br>4. N3 recibe la respuesta de N1 (RpBloque(BD)) y: (1) responde con el bloque a N2 (RpBloque(BD)) y (2) escribe BD en MP y pone el estado del bloque en el directorio a Válido<br>5. N2, cuando recibe RpBloque(BD), introduce el bloque en su cache en estado Compartido | **N0) Inválido**<br>**N1) Compartido**<br>**N2) Compartido**<br>**N3) Inválido**<br>**D) Válido**<br>V \| \| 1 \| 1 \| \| |
| **N0) Inválido**<br>**N1) Compartido**<br>**N2) Compartido**<br>**N3) Inválido**<br>**D) Válido**<br>V \| \| 1 \| 1 \| \| | P3 lee D | N3 lee BD de su propia memoria, lo introduce en su cache en estado Compartido y activa el bit de copia en la entrada de BD en el directorio de N3 | **N0) Inválido**<br>**N1) Compartido**<br>**N2) Compartido**<br>**N3) Compartido**<br>**D) Válido**<br>V \| \| 1 \| 1 \| 1 \| |
| **N0) Inválido**<br>**N1) Compartido**<br>**N2) Compartido**<br>**N3) Compartido**<br>**D) Válido**<br>V \| \| 1 \| 1 \| 1 \| | P0 escribe en D | 1. N0 envía a N3 petición de lectura de BD con acceso exclusivo PtLecEx(BD)<br>2. N3 recibe PtLecEx(BD) y, como tiene el bloque en estado Válido: (1) pone en el directorio el estado de BD en pendiente de Inválido, (2) envía los paquetes RvInv(BD) a los nodos que, según el directorio, tienen copia del bloque, (3) desactiva los bits de presencia 1, 2 y 3, activa bit de presencia 0 (de N0).<br>3. N1, N2 y N3, cuando reciben RvInv(BD): (1) invalidan su copia de BD y (2) responden a N3 confirmando la invalidación | **N0) Modificado**<br>**N1) Inválido**<br>**N2) Inválido**<br>**N3) Inválido**<br>**D) Inválido**<br>I \| 1 \| \| \| \| |

| | | | |
|---|---|---|---|
| | | RpInv(BD).<br>4. N3, cuando recibe todas las RpInv(BD): (1) envía respuesta con el bloque a N0 confirmando invalidación RpBloqueInv (espera a todas las invalidaciones para garantizar un orden en los accesos a BD y así garantizar coherencia) y (2) pone el estado de BD en el directorio a Inválido<br>5. N0 recibe RpBloqueInv de N3, escribe BD en su cache, modifica BD y lo pone en estado Modificado | |

**Ejercicio 4.** Supongamos que se va a ejecutar en paralelo el siguiente código (inicialmente *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 adelanten a las escrituras que tiene su buffer. Obsérvese que hay varios posibles resultados.

**Solución**

El compilador no altera ningún orden garantizado ya que se supone, según el enunciado, que no altera el código.

**(a)** Si P1 es el primero que imprime puede imprimir 0, 1 o 2, pero P2 podrá imprimir sólo 2. Esto es así porque se mantiene orden secuencial (el hardware parece ejecutar los accesos a memoria del código que ejecuta un procesador en el orden en el que están en dicho código) y, por tanto, cuando P1 lee "y" (instrucción 1.3 en el código, esta instrucción lee "y" para imprimir su contenido), ha asignado ya a "x" un 2 (punto 1.2 en el código) ya que esta asignación está antes en el código que la lectura de "y".

| **P1** | **P2** |
|---|---|
| `(1.1) x=1;` | `(2.1) y=1;` |
| `(1.2) x=2;` | `(2.2) y=2;` |
| `(1.3) print y ;` | `(2.3) print x ;` |

Si P2 es el primero que imprime podrá imprimir 0, 1 o 2, pero entonces P1 sólo puede imprimir 2. Esto es así porque se mantiene orden secuencial y, por tanto, cuando P2 lee "x" (punto 2.3 en el código), ha asignado ya a "y" un 2 (punto 2.2 en el código) ya que esta asignación está antes en el código que la lectura de "x" y se mantiene orden secuencial en los accesos a memoria, es decir, los accesos parecen completarse en el orden en el que se encuentran en el código.

Se puede obtener como resultado de la ejecución las combinaciones que hay en cada una de las líneas:

P1 P2
0 2 (en este caso P1 imprime 0 y P2 imprime 2)
1 2
2 2
2 0
2 1

**(b)** Si no se mantiene el orden W→R además de los resultados anteriores, los dos procesos pueden imprimir:

P1 P2
1 1 (en este caso P1 imprime 1 y P2 imprime 1)
0 1

1 0
0 0

Se pueden imprimir también las combinaciones anteriores porque no se asegura que cuando un procesador ejecute la lectura de la variable que imprime `print` (puntos 1.3 y 2.3 en los códigos) haya ejecutado las instrucciones anteriores que escriben en x (P1 en los puntos 1.1 y 1.2 del código) o en y (P2 en los puntos 2.1 y 2.2). Esto es así porque no se garantiza el orden W->R y, por tanto, una lectura puede adelantar a escrituras que estén antes en el código secuencial. P1 puede leer y (1.3) antes de escribir en x 2 (1.2) o incluso antes de escribir en x 1 (1.1). Igualmente P2 puede leer x (2.3) antes de escribir en y 2 (2.2) o antes de escribir en y 1 (2.1).

Teniendo esto en cuenta P1 puede imprimir 1 o 2 o 0, y P2 1 o 2 o 0. Todas las combinaciones son posibles.

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

| **P1** | **P2** |
|---|---|
| `x=30;` | `while (flag==0) {};` |
| `y=40;` | `r1=x;` |
| `flag=1;` | `r2=y;` |

Qué datos puede obtener P2 en 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 consistencia de liberación. Razone su respuesta.

**Solución**

El compilador no altera ningún orden garantizado ya que se supone, según el enunciado, que no altera el código.

**(a)** Si se implementa consistencia secuencial en r1 se almacena 30 y en r2 40.

RAZONAMIENTO:

Esto es así porque se garantizan los órdenes de acceso a memoria W->W y R->R que aparecen en el código que ejecuta un proceso:

1) En P1, al garantizarse el orden W->W, la escritura de 1 en flag no se realiza hasta que no se han realizado las escrituras en x e y que preceden a la escritura de flag en el código de P1.

2) Hasta que P2 no encuentra en flag un 1 no almacena en r1 el contenido de x ni en r2 el contenido de y. Al mantenerse el orden R->R, en P2 no se adelanta la lectura de x ni la lectura de y a la lectura de flag aunque se permita la lectura especulativa (ejecutar instrucciones cuya ejecución depende de una condición de salto antes de verificar que se cumple la condición).

Por tanto, como se garantiza W->W y además se garantiza R->R, si P2 ve en flag un 1 y, por tanto, sale del bucle, va a ver al leer en x 30 y en y 40.

**(b)** Si se implementa consistencia de liberación se pueden dar los siguientes resultados

| r1 | r2 |
|---|---|
| 0 | 0 |
| 0 | 40 |
| 30 | 0 |
| 30 | 40 |

RAZONAMIENTO:

1) Al no garantizarse el orden entre accesos de escritura (W->W), P1 podría escribir en flag un 1 antes de escribir 30 en x o 40 en y (obsérvese que la escritura y=40 podría también adelantar a x=30). Por lo que P2

podría leer en flag un 1 y, por tanto, salir del bucle, antes de que en x o en y pudiera ver los valores que escribe P1 en estas variables (vería entonces 0).

2) Al no garantizarse el orden entre accesos de lectura (R->R), si se permite ejecutar lecturas cuya ejecución depende de una condición de salto antes de verificar que se cumple la condición (ejecución especulativa), en P2 se podría, en la última iteración del bucle, adelantar la lectura de x o la lectura de y o ambas a la de flag. En este caso P2 podría entonces leer de forma *especulativa los valores de x e y que tienen en su cache antes de que P1 los modifique y modifique flag*. En estas circunstancias podría obtenerse en r1 o en r2 o en ambos un 0. Otras combinaciones serían también posibles.

◀

**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? Razone su respuesta. **(b)** ¿Cómo se debería implementar la función de liberación de un cerrojo simple si se usan procesadores Itanium? Razone su respuesta.

**Solución**

**(a)** La función de liberación se utiliza después de acceder a las variables compartidas para permitir que, después de un acceso por parte de un flujo de control a estas variables, otros flujos de control puedan acceder. Los procesadores de la línea x86 sólo relajan W->R. Se podría implementar puesto que en el multiprocesador no se permite que una escritura adelante a una lectura o escritura anterior; por tanto, la liberación del cerrojo no va a realizarse antes de haber terminado los accesos a las variables compartidas.

**(a)** Los Itanium implementan un modelo de ordenación que relaja todos los órdenes. No obstante, proporcionan instrucciones para garantizar órdenes apropiados cuando resulta necesario; en particular, proporcionan una escritura con liberación que garantiza que no se escribe hasta que no hayan terminado los accesos a memoria que preceden a la instrucción de liberación. Para implementar la función de liberación de un cerrojo simple bastaría usar una instrucción de almacenamiento en memoria de liberación (`st.rel`)

◀

**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 y `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.

**Solución**

No hay problemas con `Fetch_&_Or(k,1)` porque es una operación atómica que contiene R (también tiene W) y ni las R ni las W pueden adelantar a R anteriores en el orden del programa. Pero, al no garantizarse el orden W->W, la liberación del cerrojo, es decir la asignación de 0 a k puede adelantar los accesos a la variable compartida anteriores, en particular, puede adelantar a la escritura de sum o a la escritura y la lectura. Si esto ocurre más de un flujo de control puede leer el mismo valor en sum, acumular a ese valor su variable local sump y almacenar el resultado. En la variable sum acaba acumulado entonces, tras la escritura de los threads que han leído lo mismo de sum, sólo el contenido de sump de uno de esos threads, en particular, del último que ha escrito.

Para obtener los posibles resultados, primero hay que obtener lo que calcula cada thread en sump. Teniendo en cuenta que el bucle `for` asigna iteraciones consecutivas a distintos threads (turno rotatorio) el thread 0 suma 1+4+7=12, el 1 suma 2+5+8=15 y el 2 suma 3+6=9.

| | resultado | comentario |
|---|---|---|
| **Si t leen distinto valor** | 12+15+9=36 | Si no hay problemas debido a que la escritura de liberación adelante a los accesos a sum anteriores |
| **Si todos leen 0 en sum** | 12 | Si los tres flujos de control llegan a leer el valor 0 inicial de sum y el último que escribe en sum tras actualizar su valor es el thread 0 |
| | 15 | Si los tres flujos de control llegan a leer el valor 0 inicial de sum y el último que escribe en sum es el thread 1 |
| | 9 | Si los tres flujos de control llegan a leer el valor 0 inicial de sum y el último que escribe en sum es el thread 2 |
| **Si dos leen el mismo valor en sum, y el otro un valor distinto** | 12+15=27 ó 12+9=21 | - Si el flujo de control 0 ha logrado acumular primero sin problemas y el 1 y el 2 leen el valor que tiene sum tras la acumulación de 0. Si esto ocurre entonces 1 y 2 acceden al mismo valor, 12, le acumulan cada uno lo que han calculado y escriben el resultado de la acumulación en sum. El resultado será 27 si el último que escribe es 1 y 21 si el último que escribe es 2.<br>- Si los flujos 1 y 2 leen 0 los dos y acumulan su resultado parcial. Si de los dos escribe el último 1, entonces 0 sumará 12 a 15 obteniéndose 27. Si escribe el último 2, entonces 0 sumará 12 a 9, obteniéndose 21. |
| | 15+12=27 ó 15+9=24 | - Si el flujo de control 1 ha logrado acumular primero sin problemas y el 0 y el 2 acceden al valor que tiene sum justo tras la acumulación de 1. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 15, le acumulan cada uno lo que han calculado y escriben el resultado de la acumulación en sum. El resultado será 27 si el último que escribe es 0 y 24 si el último que escribe es 2.<br>- Si los flujos 0 y 2 leen 0 los dos y acumulan su resultado parcial. Si de los dos escribe el último 0, entonces 1 sumará 15 a 12 obteniéndose 27. Si escribe el último 2, entonces 1 sumará 15 a 9, obteniéndose 24. |
| | 9+12=21 ó 9+15=24 | - Si el flujo de control 2 ha logrado acumular primero sin problemas y el 0 y el 1 acceden al valor que tiene sum justo tras la acumulación de 2. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 9, le acumulan cada uno lo que han calculado y escriben el resultado de la acumulación en sum. El resultado será 21 si el último que escribe es 0 y 24 si el último que escribe es 1.<br>- Si los flujos 0 y 1 leen 0 los dos y acumulan su resultado parcial. Si de los dos escribe el último 0, entonces 2 sumará 9 a 12 obteniéndose 21. Si escribe el |

último 1, entonces 2 sumará 9 a 15, obteniéndose 24.

**(b)** En este caso la liberación del cerrojo no puede adelantar nunca a los accesos a la variable compartida `sum` porque la liberación es una escritura y no se admiten que las escrituras adelanten a accesos anteriores en el código. Por tanto, se hace el acceso a en exclusión mutua por parte de los tres flujos. El único resultado posible sería la suma de todos los componentes de la lista porque se acumula correctamente en `sum` la suma parcial calculada en `sump` por todos los flujos de control: 12+15+9=36.

◀

**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`?

### Solución

Pueden surgir problemas pudiendo incluso no funcionar bien como barrera.

Si se usa el contador global en el `if` que comprueba si contador es ya igual al número de procesos que se sincronizan con la barrera, un proceso puede encontrar, cuando llegue al `if`, que el contador es igual al número de procesos sin ser el último proceso que ha incrementado el contador. Esto puede provocar el siguiente problema:

Además del proceso que ha hecho el contador igual al número de procesos, otros que lo han incrementado antes pueden encontrar la condición del `if` verdadera y escribir, por tanto, en contador global y en la bandera global. Todas estas escrituras provocan accesos a través de la red para transferir el bloque de memoria que contiene la información sobre la barrera cuando una trasferencia por cada variable a modificar sería suficiente. Estas transferencias adicionales simplemente devalúan prestaciones, nada más. El problema aparece si la barrera se vuelve a reutilizar por los mismos procesos y el SO suspende a uno de los procesos que encuentran contador igual a `num_procesos` antes de escribir un 0 en el contador global. Puede ocurrir que, mientras este proceso está bloqueado, el resto de procesos salgan de la barrera y vuelvan a reutilizarla. Conforme los procesos van ejecutando de nuevo la barrera, incrementan el contador global y se quedan esperando a que éste llegue a ser igual a `num_procesos`. Pero nunca llegará a alcanzar este valor porque cuando el proceso suspendido vuelva a ejecutarse pondrá a 0 el contador global perdiéndose la cuenta del número de procesos que están ya esperando en la barrera.

◀

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

### Solución

En la siguiente figura se destaca en rojo el cambio realizado. Se decrementa uno a num_procesos en el if porque Fetch&Add devuelve el valor antes de la modificación, no después de la modificación. Por tanto, cuando llega el último proceso a la barrera se devuelve nom_procesos-1.

**Barrera *sense-reversing***

```
Barrera(id, num_procesos)
{
  bandera_local= !(bandera_local)       //se complementa la bandera local
  cont_local = Fecht&Add (bar[id].cont,1);        //cont_local es una variable privada
    if (cont_local ==num_procesos-1)  {
     bar[id].cont=0;                          //se hace 0 el contador asociado a la barrera
     bar[id].bandera= bandera_local;    //para liberar los procesos en espera
  }
  else  while (bar[id].bandera!= bandera_local) {};   //espera ocupada
}
```

**Ejercicio 10.** Se quiere paralelizar 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)** Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva `Fetch&Or` para garantizar exclusión mutua.

**(b)** Paralelizar el anterior ciclo en un multiprocesador que además tiene la primitiva `Fetch&Add`.

**Solución**

Se debe tener en cuenta que el único orden que no garantiza el hardware es el orden W->R.

**(a)** Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva `Fetch&Or` para garantizar exclusión mutua.

**Bucle**

```
for (i=0;i<100;i++) {
   código para i    }
```

**Dinámica**

```
while (Fetch_&_Or(k,1)==1) {}; //lock(k)
   n=i;  i=i+1;
mov k,0;  //unlock(k)
while (n<100) {
      código para n ;
      while (Fetch_&_Or(k,1)==1) {};
         n=i;  i=i+1;
      mov k,0;  //unlock(k)
```

Nota: la variable i estaría inicializada a 0 (por ejemplo, se puede iniciar cuando se declara)

Se supone que el compilador no cambia de sitio "`mov k,0`".

**(b)** Paralelizar el anterior ciclo en un multiprocesador que además tiene la primitiva `Fetch&Add`.

**Bucle**

```
for (i=0;i<100;i++) {
   código para i    }
```

**Dinámica**

```
n=Fetch_&_Add(i,1);
while (n<100) {
      código para n ;
      n=Fetch_&_Add(i,1);
 }
```

Nota: la variable i estaría inicializada a 0 (por ejemplo, se puede iniciar cuando se declara)

**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;
  k=0;
  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 de memoria de ordenación débil? Razone por qué.

### Solución

```
                        Barrera(id, num_procesos)
                        {
                          band_local= !(band_local)
        (R,W) atómica     while (fetch_&_or(k,1)==1) {};
  R seguida de W (RAW)      cont_local = ++bar[id].cont;
                  W       k=0;
                          if (cont_local == num_procesos)  {
                               bar[id].cont=0;
                               bar[id].band=band_local;
                          }
                          else while (bar[id].band != band_local) {};
                        }
```

Habría problemas si dos o más threads pueden estar al mismo tiempo ejecutando accesos a variables compartidas de la sección crítica; es decir, en este caso, accediendo a la variable compartida `bar[id].cont`. Esto podría ocurrir si:

1. La apertura del cerrojo o escritura en k de 0 adelanta a los accesos anteriores a la variable compartida `bar[id].cont`, porque un proceso podría encontrar el cerrojo abierto estando otro proceso aún en la sección crítica, o
2. El acceso a la variable compartida (lectura) adelanta a la operación atómica de escritura y lectura `fetch_&_or`, porque el proceso que la adelanta accedería a la variable compartida pudiendo estar otro proceso accediendo a ella. El enunciado del ejercicio dice que esto no puede ocurrir.

**(a)** Sí, funciona bien como barrera. La escritura en k de 0 no puede adelantar a los accesos anteriores a la variable compartida `bar[id].cont`, puesto que la arquitectura no admite que una escritura pueda adelantar a lecturas o escrituras anteriores en el orden del programa. Por tanto, un thread asigna un 0 a k cuando ya ha escrito en la variable compartida.

**(b)** No, no funciona bien como barrera. La escritura en k de 0 puede adelantar a los accesos anteriores a la variable compartida `bar[id].cont`, puesto que la arquitectura permite que una escritura pueda adelantar a lecturas o escrituras anteriores en el orden del programa. Por tanto, un thread puede asignar un 0 a k cuando aún no ha escrito en la variable compartida y, por tanto, otro thread podría entrar en la sección crítica por encontrarse el cerrojo abierto y podría leer el mismo valor de `bar[id].cont` que el thread que le abrió el cerrojo.

**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`, `medl` y `varil` son variables locales; `i`, `med`, `vari`, el vector `x` y `N` son variables compartidas; inicialmente `med`, `vari`, `medl` y `varil` son 0.

```
(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
(5)  med = med + medl/N; vari = vari + varil/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 varias hebras 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 puntuará según 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 puntuará según prestaciones). Indique qué variables son compartidas y cuáles locales.

NOTA: En todos los apartados puede añadir o quitar variables si lo estima conveniente.

### Solución

**(a)** Hay dos tipos de errores en el código:

1. No se accede en exclusión mutua a las variables compartidas `med` y `vari` por parte de los diferentes threads (línea de código 5). Esto permite que puedan intentar varios threads a la vez acumular el resultado parcial que han calculado (carrera). Por ejemplo, varias pueden cargar el mismo valor en `med`, acumular a ese valor su variable local `medl` y almacenar el resultado en `med`. El resultado de `med` acumulado en `med` será entonces el cargado por todas ellas más el contenido de `medl` de sólo una de ellas, de la última que ha almacenado. Por lo que no se acumularía lo calculado por todas ellas.
2. Las operaciones de la línea (6) leen y modifican la variable compartida `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 `varil` (e imprimir esta variable de 0 en el `printf`) o meter (6) dentro del `if` que acompaña al `printf`
3. El thread 0 obtiene el valor definitivo de `vari` a partir de `med` y `vari` (línea de código 6) e imprime (línea de código 7) sin esperar a que todos los threads acumulen en las variables compartidas `med` y `vari` los resultados parciales que han obtenido en el bucle en las variables locales `medl` y `varil`. Esto supone que cuando imprime pueden haber intentado acumular su resultado parcial el thread 0 y una combinación del resto de threads en un número de 0 a `nthread-1`. Por este motivo, aunque se accediera en exclusión mutua a las variables compartidas, se podrían imprimir distintos resultados en distintas ejecuciones.

**(b)** Se accederá en exclusión mutua a `med` y `vari` implementando un cerrojo simple con una variable compartida `k1` (ver código en calabaza), se tiene que añadir una barrera antes de que imprima el thread 0 (ver código en azul) y se introduce (6) dentro del if. La barrera se debe implementar también usando un cerrojo simple (`k2`):

```
(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }

(5)  medl=medl/N; varil=varil/N;
     while (test_&_set(k1)) {};   //lock(k1)
     med = med + medl; vari = vari + varil;
     k1=0;                        //unlock(k1)

     bandera_local= !(bandera_local)//se complementa la bandera local
     while (test_&_set(k2)) {};     //lock(k2)
     bar[id].cont+=1; cont_local = bar[id].cont;   //cont_local es local
     k2=0;                          //unlock(k2)
     if (cont_local ==num_procesos)  {
            bar[id].cont=0;    //se hace 0 el contador asociado a la barrera
            bar[id].bandera= bandera_local;  // libera procesos en espera
     }
     else  while (bar[id].bandera!= bandera_local) {};

(6)  if (ithread==0) { vari= vari - med*med;
(7)                    printf("varianza = %f", vari);} //imprime en pantalla
```

`k1`, `k2`, `bandera_local`, `cont_local`, `ithread`, `i`, `medl` y `varil` son variables locales; el vector bar, `med`, `vari`, el vector `x` y `N` son variables compartidas; inicialmente `k1`, `k2`, `med`, `vari`, `medl` y `varil` son 0.

**(c)** Con `fetch_&_add()`, para el acceso en exclusión mutua no es necesario usar variables compartidas extras como cerrojos:

```
(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
(5)  fetch_&_add(med,medl/N); fetch_&_add(vari,varl/N);

     bandera_local= !(bandera_local)//se complementa la bandera local
     cont_local = fetch_&_add(bar[id].cont,1);    //cont_local es local
     if (cont_local==num_procesos-1)  {
            bar[id].cont=0;    //se hace 0 el contador asociado a la barrera
            bar[id].bandera= bandera_local;  // libera procesos en espera
     }
     else  while (bar[id].bandera!= bandera_local) {};

(6)  if (ithread==0) { vari= vari - med*med;
(7)                    printf("varianza = %f", vari);} //imprime en pantalla
```

`bandera_local`, `cont_local`, `ithread`, `i`, `medl` y `varil` son variables locales; el vector bar, `med`, `vari`, el vector `x` y `N` son variables compartidas; inicialmente `med`, `vari`, `medl` y `varil` son 0.

**(d)** Con `compare_&_swap()`, para el acceso en exclusión mutua no es necesario usar variables compartidas extras como cerrojos:

```
(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
(5)
     medl=medl/N; varil=varil/N;
     do
        a = med;
        b = a + medl;   //a y b son variables locales
        compare&swap(a,b,med);
     while (a!=b);
     do
        a = vari;
        b = a + varil;
        compare&swap(a,b,vari);
     while (a!=b);

     bandera_local= !(bandera_local)//se complementa la 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;    //se hace 0 el contador asociado a la barrera
           bar[id].bandera= bandera_local;  // libera procesos en espera
     }
     else  while (bar[id].bandera!= bandera_local) {};

(6)  if (ithread==0) { vari= vari - med*med;
(7)                    printf("varianza = %f", vari);} //imprime en pantalla
```

a, b, bandera_local, cont_local, ithread, i, medl y varil son variables locales; el vector bar, med, vari, el vector x y N son variables compartidas; inicialmente med, vari, medl y varil son 0.

**Ejercicio 13.** Se ha extraído la siguiente implementación de cerrojo (spin-lock) para x86 del kernel de Linux (http://lxr.free-electrons.com/source/arch/x86/include/asm/spinlock.h):

```
typedef struct {
    unsigned int slock;
 } raw_spinlock_t;
…
/*Para un número de procesadores menor que 256=2^8
-#if (NR_CPUS < 256)
…
-static __always_inline void __ticket_spin_lock(raw_spinlock_t *lock)
-{
-   short inc = 0x0100;
-
-   asm volatile (
-      "lock  xaddw %w0, %1\n"  /*w: se queda con los 16 bits menos significativos*/
-      "1:                \t"    /*b: se queda con el byte menos significativo*/
-      "cmpb %h0, %b0     \n\t"  /*h: coge el byte que sigue al menos significativo*/
-      "je 2f             \n\t"  /*f: forward */
-      "rep ; nop         \n\t"  /*retardo, es equivalente a pause*/
-      "movb %1, %b0      \n\t"
-      /* don't need lfence here, because loads are in-order */
-      "jmp 1b            \n"    /*b: backward */
-      "2:"
```

```
-      : "+Q" (inc), "+m" (lock->slock) /*%0 es inc, %1 es lock->slock */
- /*Q asigna cualquier registro al que se pueda acceder con rh: a, b, c y d; ej. ah, bh …
*/
-      :
-      : "memory", "cc");
-}
-
-static __always_inline void __ticket_spin_unlock(raw_spinlock_t *lock)
-{
-   asm volatile( “incb %0"   /*%0 es lock->slock */
-           : "+m" (lock->slock)
-           :
-           : "memory", "cc");

-}
```

Conteste a las siguientes preguntas:

**(a)** Utiliza una implementación de cerrojo con etiquetas ¿Cuál es el contador de adquisición y cuál es el contador de liberación?

**(b)** Describa qué hace `xaddw %w0, %1` ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?

**(c)** Describa qué hace `cmpb %h0, %b0` ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?

**(d)** ¿Por qué cree que se usa el prefijo `lock` delante de la instrucción `xaddw`?

NOTAS: (1) Puede consultar las instrucciones en el manual de Intel con el repertorio de instrucciones (Volumen 2 o volúmenes 2A, 2B y 2C) que puede encontrar aquí http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html .

(2) Si no recuerda la interfaz entre C/C++ y ensamblador en gcc (se ha presentado en Estructura de Computadores), consulte el manual de gcc aquí http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Extended-Asm.html#Extended-Asm (http://gcc.gnu.org/onlinedocs/)

### Solución

**(b)** `lock->slock` contiene el contador de liberación en los bits de 0 a 7 liberación (`lock->slock[7...0]`) y el de adquisición en los bits de 8 a 15 (`lock->slock[15...8]`).

**(c)** `xaddw %w0, %1` almacena en los 16 bits menos significativos del registro al que se ha asigna `inc` (%0) los 16 bits (sufijo w) menos significativos de `lock->slock` (%1) y asigna a `lock->slock` (contador de adquisición y contador de liberación) el resultado de sumarlo con `inc`. Como consecuencia: **(1)** incrementa en uno el contador de adquisición (`lock->slock[15...8]`) dado que `inc` tiene un 1 en el bit 8 (`inc` contiene 0x0100) y **(2)** almacena en `inc[15...8]` (%h0) el valor de este contador antes de la modificación y en `inc[7...0]` (%b0) el valor del contador de liberación (`lock->slock[7...0]`).

**(d)** `cmpb %h0, %b0` compara el valor actual del contador de liberación `inc[7...0]` (%b0) y el de adquisición `inc[15...8]` (%h0); es decir, resta ambos contadores modificando sólo el registro de estado. En las instrucciones posteriores se usa el resultado de la comparación (los bits de estado resultantes de la comparación). Si son iguales ambos contadores (bit z del registro de estado a 1), abandona la función `lock` del cerrojo, y si son distintos actualiza el valor del contador de liberación cargando lo que hay en `lock->slock[7...0]` en `inc[7...0]` (%b0)

**(e)** Se requiere el prefijo `lock` para que la lectura y escritura en memoria que realiza la instrucción `xaddw` se hagan de forma atómica. Si `xaddw` no fuese atómica dos flujos de control podrían leer el mismo valor del contador de adquisición y, como consecuencia, más de un flujo podría entrar a la vez en una sección crítica.

## 2 Cuestiones

**Cuestión 1.** Diferencias entre núcleos con multithread temporal y núcleos con multithread simultánea.

**Solución**

Un núcleo con multithread simultánea es un núcleo superescalar que puede emitir a unidades funcionales instrucciones de múltiples threads distintos por lo que puede ejecutar operaciones de distintos threads en paralelo. Mientras que un núcleo con multithread temporal es un núcleo segmentado, superescalar o VLIW que ejecuta varios threads concurrentemente, pero no en paralelo (sólo emite instrucciones de un thread a unidades funcionales), es decir, multiplexando en el tiempo el uso de las etapas del cauce entre los threads. Para la multiplexación tiene hardware que se encarga de conmutar entre threads.

◀

**Cuestión 2.** Diferencias entre núcleos con multithread temporal de grano fino y núcleos con multithread temporal de grano grueso.

**Solución**

En un FGMT el hardware puede conmutar de thread cada ciclo de reloj, con una penalización de 0 ciclos, por turno rotatorio o por un evento (dependencia funcional, fallo en cache de nivel 1, operación con CPI de varios ciclos, salto no predecible) combinado con alguna técnica de planificación (p. ej. Thread que lleva menos sin ejecutarse), mientras que en un CGMT se conmuta tras varios ciclos de reloj, con una penalización que puede ser distinta de 0, tras un nº de ciclos prestablecido o por un evento estático (ejecución de una instrucción añadida al repertorio o ya incorporada -carga/almacenamiento, salto) o dinámico (como el fallo de la cache de último nivel o una interrupción).

◀

**Cuestión 3.** Suponga un multiprocesador con protocolo MESI de espionaje. Si un controlador de cache observa en el bus un paquete de petición de lectura exclusiva de un bloque que tiene en estado S, debe (indique cuál sería la respuesta correcta y razone por qué es la respuesta correcta):

a) Generar un paquete de respuesta con el bloque y pasar el bloque a estado I.
b) Pasar el bloque a estado I.
c) Generar un paquete de respuesta con el bloque y pasar el bloque a estado E.
d) No tiene que hacer nada

**Solución**

(b) Pasar el bloque a estado I.
Al estar su copia del bloque en estado Compartido no necesita entregar copia del bloque, porque lo tiene válido la memoria y será ella quien generará un paquete de respuesta con el bloque. Al ser la petición de acceso exclusivo al bloque, significa que quien ha enviado el paquete va a escribir en el bloque, por tanto, la copia que tiene el nodo ya no será válida, de ahí que pase a estado Inválido.

◀

**Cuestión 4.** Suponga un multiprocesador con protocolo MESI de espionaje. Si un nodo observa en el bus un paquete de petición de lectura exclusiva de un bloque que tiene en estado M, ¿qué debe hacer? Razone su respuesta.

**Solución**

(1) Generar un paquete de respuesta con el bloque, porque el paquete de petición incluye lectura del bloque y la memoria no tiene copia válida, él tiene la única copia válida en todo el sistema.

(2) Pasar el bloque a estado I, porque si se solicita un acceso exclusivo al bloque es porque quien ha envido el paquete va a escribir en su cache en la copia que va a recibir, por tanto, la copia que está en estado M ya no estará actualizada, será inválida.

◀

**Cuestión 5.** Suponga un multiprocesador con el protocolo MESI de espionaje. Si el procesador de un nodo escribe en un bloque que tiene en su cache en estado I, debe (indique cuál sería la respuesta correcta y razone por qué es la respuesta correcta):

a) Generar paquete de petición de acceso Exclusivo al bloque y pasar el bloque a M
b) Generar paquete de petición de acceso Exclusivo al bloque y pasar el bloque a estado E
c) Generar paquete de petición de acceso Exclusivo al bloque con lectura y pasar el bloque a estado E
d) Generar paquete de petición de acceso Exclusivo al bloque con lectura y pasar el bloque a M

**Solución**

(d) Generar paquete de petición de acceso Exclusivo al bloque con lectura y pasar el bloque a M

Razonamiento:

(1) Paquete con lectura: como la cache del nodo no tiene copia válida del bloque (debido al estado inválido), el controlador de cache del nodo tiene que generar un paquete que incluya lectura.

(2) Paquete con acceso exclusivo: como se va a escribir en la copia que se reciba del bloque, el controlador de cache del nodo tiene que pedir acceso exclusivo para que las copias del bloque en otras caches sean invalidadas. Esto es necesario porque las siguientes lecturas del bloque que se realicen en otros nodos deberán acceder a la última modificación del mismo que es la que se va a realizar en este nodo en lugar de acceder a copias no actualizadas del bloque que estén en sus caches.

(3) Pasar a estado modificado: como el nodo va a escribir en el bloque cuando lo reciba será la única copia válida del bloque en el sistema, por tanto, deberá estar en estado modificado para que el controlador de cache sepa que debe responder con el bloque si se recibe alguna petición que incluya lectura del bloque.

.

◀

**Cuestión 6.** ¿Cuál de los siguientes modelos de consistencia permite mejores tiempos de ejecución? Justifique su respuesta.

a) modelo de ordenación débil
b) modelo implementado en los procesadores de la línea x86
c) modelo de consistencia secuencial
d) modelo de consistencia de liberación

**Solución**

(d) modelo de consistencia de liberación

Tanto el modelo de ordenación débil como el de consistencia de liberación relajan todos los órdenes entre operaciones de acceso a memoria (W->R, W->W, R->W,R) por lo que permitirán mejores tiempos de ejecución que el resto porque ningún acceso tiene que esperar a otro.

Pero el de liberación ofrece mejores prestaciones que el de ordenación débil porque:

(1) El de ordenación débil ofrece instrucciones máquina que permiten que, si S es una operación de sincronización, se garanticen los siguientes órdenes:

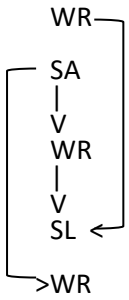
- S->WR (hasta que no termina la operación de sincronización no puede empezar ningún acceso a memoria posterior)
- WR->S (hasta que no terminen los accesos a memoria que hay antes de una operación de sincronización no puede empezar la operación de sincronización)

Con estos órdenes los accesos a memoria que hay detrás de una sincronización no pueden empezar hasta que no hayan acabado todos los accesos que hay delante de la sincronización. Por tanto, se garantiza el siguiente orden entre operaciones de acceso a memoria y operaciones de sincronización de adquisición SA y de liberación SL: WR->SA->WR->SL->WR. No hay ningún tipo de paralelismo entre operaciones de acceso a memoria antes y después de operaciones de sincronización

(2) Mientras que el modelo de liberación ofrece instrucciones máquina menos restrictivas que permiten garantizar los siguientes órdenes entre accesos a memoria de lectura L y escritura W y operaciones de sincronización de adquisición SA y liberación SL:

- SA->WR (hasta que no termina la operación de sincronización de adquisición no puede empezar ningún acceso a memoria posterior)
- WR->SL (hasta que no terminen los accesos a memoria que hay antes de una operación de sincronización de liberación no puede empezar la operación de sincronización de liberación)

Con estos órdenes los accesos a memoria que hay detrás de una sincronización de liberación pueden empezar aunque no hayan terminado los que hay delante, y los que hay delante de una operación de adquisición pueden terminar después de las que hay detrás. O sea, las operaciones de acceso a memoria que hay antes de la operación de adquisición se pueden realizar en paralelo a las operaciones que hay detrás de la adquisición y antes de la liberación, y las operaciones de acceso a memoria que hay detrás de la operación de liberación se pueden realizar en paralelo a las que hay entre la operación de adquisición y la de liberación:




**Cuestión 7.** Indique qué expresión no se corresponde con la serie (justifique su respuesta):

a) `Lock`
b) `Fetch_and_Or`
c) `Compare_and_Swap`
d) `Test_and_Set`

**Solución**

Hay una función software de cerrojos, `lock`, y tres instrucciones máquina utilizadas para mejorar prestaciones en la sincronización de flujos de control: `Fetch_and_Or`, `Compare_and_Swap` y `Test_and_Set`.

Luego `lock` no se corresponde con la serie.