

6. Problemas

Problema 1. En un multiprocesador SMP con 4 procesadores o nodos (N0, N1, N2, N3) basado en un bus y que implementa el protocolo MESI para mantener la coherencia, se produce la siguiente secuencia de accesos a memoria a una dirección D que no se encuentra en ninguna caché:

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

Indicar los estados en las cachés del bloque en el que se encuentra D y las acciones que se producen en el sistema para cada uno de estos eventos.

Solución

Datos del ejercicio

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

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

Se va a utilizar una tabla para describir las acciones generadas para cada evento teniendo en cuenta el estado del bloque. En la tabla se usará una fila para cada uno de los 5 eventos.

Hay 4 nodos con caché (N0 a N3). Intervienen N1, N2 y N3. Se van a utilizar las siguientes 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). Por tanto, en los casos en los que se generaría este paquete, el controlador de caché genera un paquete de petición de acceso exclusivo con lectura (PtLecEx).

ESTADO INICIAL	EVENTO	ACCIÓN	L1) Inválido N1) Inválido N2) Inválido N3) Inválido	L1) Exclusivo N1) Exclusivo N2) Inválido N3) Inválido
			L1) Exclusivo N1) Exclusivo N2) Compartido N3) Inválido	L1) Compartido N1) Compartido N2) Compartido N3) Inválido
	1. P1 lee k	<p>1. N1 (el controlador de caché 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.</p> <p>2. MP (el controlador de memoria de MP), al observar $PtLec(k)$ en el bus, genera la respuesta con el bloque ($RpBloque(k)$).</p> <p>3. N1 recoge del bus la respuesta depositada por la memoria principal ($RpBloque(k)$), el bloque entra en la caché de N1 en estado Exclusivo ya que no hay copia del bloque en otra caché.</p>	N1) Exclusivo N2) Inválido N3) Inválido	N1) Exclusivo N2) Inválido N3) Inválido
	2. P2 lee k	<p>1. N2 genera y deposita en el bus una $PtLec(k)$ porque no tiene k en su caché en estado válido.</p> <p>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 cachés). MP, al observar $PtLec(k)$ en el bus, genera la respuesta con el bloque ($RpBloque(k)$).</p> <p>3. N2 recoge $RpBloque(k)$ que ha depositado la memoria, el bloque entra en estado Compartido en la caché de N2 (porque hay copia del bloque en otra caché).</p>	N1) Compartido N2) Compartido N3) Inválido	N1) Compartido N2) Compartido N3) Inválido
	3. P1 escribe en k	<p>1. N1 genera petición de lectura del bloque k con acceso exclusivo ($PtLecEx(k)$) (suponemos que no hay petición de acceso exclusivo sin lectura, no hay $PtEx$).</p> <p>2. N2 observa $PtLecEx(k)$ y, como la petición incluye acceso exclusivo (Ex) a un bloque que tiene en su caché válido (Compartido), pasa su copia a estado Inválido. MP genera $RpBloque(k)$ porque observa en el bus una petición de k con lectura (Lec). N1 no recoge $RpBloque(k)$ depositada por la memoria porque tiene el bloque válido.</p> <p>3. N1 modifica la copia de k que tiene en su caché y lo pasa a estado Modificado.</p>	N1) Modificado N2) Inválido N3) Inválido	

ESTADO INICIAL	EVENTO	ACCIÓN	ESTADO SIGUIENTE
N1) Modificado N2) Inválido N3) Inválido	4. P2 escribe en k	1. N2 genera petición de lectura de k con acceso exclusivo (PtLecEx(k)) 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. 3. N2 recoge RpBloque(k), introduce k en su caché, lo modifica y lo pone en estado Modificado	N1) Inválido N2) Modificado N3) Inválido
N1) Inválido N2) Modificado N3) Inválido	5. P3 escribe en k	1. N3 genera petición de lectura con acceso exclusivo de k PtLecEx(k) 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. 3. N3 recoge RpBloque(k), introduce el k en su caché, lo modifica y lo pone en estado Modificado	N1) Inválido N2) Inválido N3) Modificado

Problema 2. Para un multiprocesador de memoria distribuida se quiere implementar un protocolo para mantenimiento de coherencia basado en directorios. Suponiendo que se necesitan dos bit de estado para un bloque en el directorio de memoria principal y que el tamaño de una línea de caché es de 64 bytes, calcular el porcentaje del tamaño de memoria principal que supone el tamaño del directorio de vector de bits de presencia para (a) un multiprocesador con 16 nodos con caché y (b) para un multiprocesador con 256 nodos.

Solución

Datos del ejercicio:

- ✓ Tamaño Línea de Caché (bloque de memoria): 64 Bytes = TLC
- ✓ 2 bit de estado por bloque en el directorio

(a) Porcentaje del tamaño de la memoria que supone el tamaño del directorio para un multiprocesador con 16 nodos con caché.

$$\text{Tamaño_directorio} = \text{No_de_bloques_memoria} \times \text{Espacio_por_bloque_en_directorio}$$

$$\text{Nº_de_bloques_memoria} = \frac{TM}{TLC} \quad (\text{TM: Tamaño Memoria principal})$$

Espacio por bloque en directorio = 18 bits (teniendo en cuenta que hay 16 nodos y 2 bits de estado)

$$\text{Tamaño - directorio} = \frac{TM}{TLC} \times (16b + 2b)$$

$$\% - de - TM = \frac{\frac{TM}{TLC} \times 18b}{TM} \times 100 = \frac{18b}{TLC} \times 100 = \frac{18b}{64B \times 8b/B} \times 100 \approx 3,5\%$$

El directorio de memoria no ocupa en este caso un porcentaje elevado del tamaño de la memoria principal. La implementación de protocolos de mantenimiento de coherencia con vector de bits de presencia es aceptable para multiprocesadores CC-NUMA con un número pequeño de nodos como los que pueda haber en implementaciones en una placa o en un chip.

(b) Porcentaje del tamaño de la memoria que supone el tamaño del directorio para un multiprocesador con 256 nodos con caché.

Espacio por bloque en directorio = 258 bits (teniendo en cuenta que hay 256 nodos y 2 bits de estado)

$$\text{Tamaño - directorio} = \frac{TM}{TLC} \times (256b + 2b)$$

$$\% - de - TM = \frac{\frac{TM}{TLC} \times 258b}{TM} \times 100 = \frac{258b}{TLC} \times 100 = \frac{258b}{64B \times 8b/B} \times 100 = \frac{258}{512} \times 100 \approx 50,4\%$$

El directorio de memoria ocuparía algo más de la mitad en este caso y superaría el tamaño de la memoria con 512 nodos. Para un número de nodos elevado no se puede utilizar un vector de bits de presencia para controlar las cachés con copia de un bloque debido al tamaño que supondría el directorio. Se pueden ver otras alternativas de implementación para grandes multiprocesadores en Julio Ortega, Mancia Anguita, 2015.

Problema 3. Se tiene un multiprocesador CC-NUMA con protocolo MSI basado en directorio distribuido (sin difusión) que usa vector de bits de presencia. Para un bloque B que se encuentra en memoria principal en estado Inválido, indicar:

- (a) Cuál sería el contenido de la entrada del directorio para ese bloque en esta situación.
Razone su respuesta.

- (b) Las transiciones de estados (en caché y en el directorio), la secuencia de paquetes generados por el protocolo de coherencia y el contenido al que pasa la entrada del directorio para ese bloque si un procesador que no tiene el bloque en su caché y no está en el nodo origen del bloque escribe en una dirección de dicho bloque. Razones su respuesta.

Solución

(a) Si el bloque está inválido en memoria principal habrá una copia del bloque válido en un única caché. El contenido en la entrada del bloque del directorio en esa situación tendrá en Inválido el estado del bloque en memoria y, en su vector de bits de presencia, tendrá un único bit activo, el resto estarán a 0. El bit activo será el que corresponda a la caché que tiene la única copia válida del bloque en el sistema:

Estado	Vector de bits de presencia							
I	0	...	0	1	0	...	0	

- (b) Si un procesador de un nodo que no tiene el bloque en su caché y no está en el nodo origen del bloque escribe en una dirección de dicho bloque:

1. Se produce un fallo de escritura que provoca la emisión desde el nodo del procesador que escribe, o nodo solicitante (S), al nodo origen del bloque (O) de un paquete de petición de lectura con acceso exclusivo al mismo PtLecEx(B).
2. El nodo origen (O) al recibir este paquete, como tiene el bloque Inválido, reenvía la petición al nodo propietario (P) del bloque RvLecEx(B). Obtiene el nodo propietario del directorio, el bit activo de la entrada del directorio para el bloque identifica al único nodo P. Antes de reenviar la petición pone el estado del bloque en memoria a Pendiente de inválido. El nodo O no atiende peticiones de un bloque en estado pendiente, así se garantiza coherencia.
3. El nodo propietario (P), al recibir la petición de lectura y acceso exclusivo a B, RvLecEx(B), invalida la copia que tiene del bloque (porque pide acceso exclusivo) y responde con el bloque (porque pide lectura) al origen confirmando acceso exclusivo. Por tanto, envía al origen un paquete de respuesta con el bloque confirmando la invalidación de la copia del bloque en su caché, RpBloqueInv(B).
4. El nodo O, cuando recibe la respuesta la envía al nodo solicitante RpBloqueInv(B). Antes del envío, deja activo en el vector de bits de presencia del bloque en el directorio sólo el bit del nodo S y pasa el estado del bloque a Inválido, porque lo va a modificar S.
5. El nodo S, cuando recibe el paquete con el bloque, RpBloqueInv (B), lo introduce en su caché, lo modifica y pasa su estado a Modificado en la entrada del bloque en el directorio de su caché (no confundir con el directorio de memoria principal).

Estado inicial	Evento	Estado final	Diagrama de paquetes
O-Memoria) Inválido S) Inválido P) Modificado Acceso remoto	Fallo de escritura	O-Memoria) Inválido S) Modificado P) Inválido	<pre> graph LR S[S] -- "1. PtLecEx" --> O[O] P[P] -- "2. RyLecEx" --> O O -- "3. RpBloqueInv" --> P O -- "4. RpBloqueInv" --> S </pre>

Problema 4. Contestar el apartado (b) del ejercicio anterior suponiendo que en el CC-NUMA se implementa un protocolo MSI con difusión.

Solución

Si un procesador de un nodo que no tiene el bloque en su caché y no está en el nodo origen del bloque escribe en una dirección de dicho bloque:

1. Se produce un fallo de escritura que provoca la difusión desde el nodo del procesador que escribe, o nodo solicitante (S), a todos los nodos N (tengan o no el bloque) de un paquete de petición de lectura con acceso exclusivo al bloque PtLecEx(B).
2. El único nodo propietario (P), al recibir la petición de lectura y acceso exclusivo a B, PtLecEx(B), invalida la copia que tiene del bloque (porque pide acceso exclusivo) y responde con el bloque (porque pide lectura) al origen confirmando acceso exclusivo, es decir, confirma la invalidación del bloque en su caché. Por tanto, envía al nodo origen (O) un paquete de respuesta con el bloque confirmando la invalidación de la copia del bloque en su caché, RpBloqueInv(B). El resto de nodos envía a O confirmación de invalidación RpInv(B). El nodo origen del bloque: (1) al ver que la petición solicita acceso exclusivo, espera recibir confirmación de invalidación de todos los cachés y (2) al ver que se pide lectura del bloque y que tiene el bloque solicitado en estado Inválido, sabe que recibirá el bloque de alguna caché. Mientras recibe las respuestas dejará el estado del bloque en Pendiente de inválido.
3. El nodo origen (O), cuando recibe todas las respuestas, envía RpBloqueInv(B) al nodo S. Antes del envío pasa el estado del bloque a Inválido, porque lo va a modificar el nodo S.
4. El nodo S, cuando recibe el paquete con el bloque, RpBloqueInv (B), lo introduce en su caché, lo modifica y pasa su estado a Modificado en la entrada del bloque en el directorio de su caché.

Estado inicial	Evento	Estado final	Diagrama de paquetes
O-Memoria) Inválido S) Inválido P) Modificado Acceso remoto	Fallo de escritura	O-Memoria) Inválido S) Modificado P) Inválido	

Problema 5. Se dispone de un multiprocesador CC-NUMA con 4 procesadores o nodos (N0-N3) y una memoria de 64 GBytes direccionada por bytes (16 GBytes por nodo). Para mantener la coherencia de caché, el multiprocesador implementa el protocolo MSI basado en directorios distribuidos (con vector de bits de presencia) y sin difusión. Cada procesador dispone de una caché de datos de último nivel de 8 MBytes con marcos de 32 bytes. En el multiprocesador se están ejecutando en paralelo cuatro threads (en N0, N1, N2 y N3) que acceden a dos vectores X[] e Y[] de 32 elementos cada uno, de 32 bits. Los vectores se encuentran almacenados a partir de una dirección de memoria múltiplo de 32: primero están almacenados los componentes de X[], y justo a continuación, los elementos de Y[]. Conteste a las siguientes preguntas:

- (a) ¿Cuál es el tamaño del subdirectorio de memoria principal de un nodo? (considerar que se almacenan estados estables y estados pendientes de un estado estable)
- (b) ¿Cuántos bloques de memoria ocupan los vectores X[] e Y[]?
- (c) Suponiendo que inicialmente los bloques que contienen ambos vectores no están en ninguna caché, ¿cuál será entonces inicialmente el contenido de las entradas del directorio de memoria principal para cada uno de estos bloques?
- (d) Indicar los estados estables de los bloques en las cachés y los cambios en los contenidos del directorio de memoria principal ante la siguiente secuencia de eventos (considere que inicialmente los bloques que contienen ambos vectores no están en ninguna caché):
 1. Lectura generada por el procesador 0 a X[0]
 2. Escritura generada por el procesador 0 a Y[0]
 3. Lectura generada por el procesador 1 a X[1]
 4. Lectura generada por el procesador 2 a X[2]
 5. Escritura generada por el procesador 2 a Y[2]
 6. Escritura generada por el procesador 1 a Y[1]

NOTA: Suponga que bloques distintos se almacenan en la caché de cada procesador en marcos de bloque diferentes.

Solución

(a) Datos:

- ✓ Tamaño memoria de un nodo (TMN) = 16 GB
- ✓ Tamaño bloque memoria y tamaño de la línea de caché (TLC) = 32 B
- ✓ N° de bits para almacenar el estado de un bloque en memoria según el protocolo MSI = 1 (para almacenar estados estables Válido o Inválido) + 1 (para indicar si está pendiente de un estado estable)

Tamaño subdirectorio de un nodo (TSN) = número de entradas en el directorio × número de bits de cada entrada = número de bloques de la memoria del nodo × (número de nodos + bits de estado) bits

$$TSN = \frac{\frac{TMN}{TLC}}{2^7} \times (4 + 2) b = \frac{\frac{16\text{ GB}}{32\text{ B}}}{2^5\text{ B}} \times 6 b = \frac{2^{34}\text{ B}}{2^5\text{ B}} \times 6 b = 2^{30} \times 3 b = \frac{2^{30} \times 3 b}{2^3 b/B} = 2^{27} \times 3 B = 2^7 \times 3 \times (2^{20} B) = 128 \times 3 MB = 384 MB$$

(b) Datos:

- ✓ Los vectores X [] e Y [] tienen 32 elementos de 32 bits cada uno, es decir, 32 elementos de 4 Bytes cada uno.
- ✓ Tamaño del bloque memoria y tamaño línea de caché (TLC) = 32 B
- ✓ X [] está almacenado en una dirección múltiplo de 32 e Y [] se encuentra justo detrás
- ✓ La memoria se dirección a nivel de byte (una dirección es un byte)

Total bloques de X [] + Y []:

$$\begin{aligned} \text{Total bloques} &= \frac{2 \text{ vectores} \times 32 \text{ comp./vector} \times 4 \text{ B/comp.}}{32 \text{ B/bloque}} \\ &= 8 \text{ (4 bloques cada uno de los vectores)} \end{aligned}$$

(c) Los bits de una entrada son 6 (como se ha indicado en (a)): 4 bits de presencia en caché (uno por cada nodo) + 2 bits de estado.

Para los 4 bloques el contenido del directorio será el siguiente: los 4 bits de presencia a 0, porque no hay copia válida en ninguna caché, el bit de estado estable a 1 (es decir a Válido) por estar válido en la memoria (ya que no hay ninguna caché que haya escrito en el bloque) y el bit de estado pendiente a 0 (por no estar pendiente de un estado estable).

(d) Sólo intervienen en la secuencia el primer bloque de X [] (se notará por B0, contiene las 8 primeros componentes X [0]X[1]X[2]X[3]X[4]X[5]X[6]X[7]) y el primero de Y [] (se notará por B4, contiene las 8 primeros componentes Y [0]Y[1]Y[2]Y[3]Y[4]Y[5]Y[6]Y[7]). El estado estable en caché puede ser Modificado (M), Compartido (S) o Inválido (I), y los estados estables en el directorio (memoria) Válido

(V) o Inválido (I). La entrada del directorio de un bloque contiene el estado en memoria del bloque (V o I), y los bits de presencia del bloque en los nodos N0, N1, N2 y N3 (que podrán estar a 0 o 1).

Situación	Contenido directorio entradas 1 ^{er} bloque de X [] (B0) y 1 ^{er} bloque de Y [] (B4)	Estado en cachés	Evento
Inicial	V 0 0 0 0 Y 0 0 0 0	x-B0 y-B4 Inválido todos los bloques en todas las cachés N0-3(B0):I ; N0-3(B4):I	1. N0 lee X[0]
	V 1 0 0 0 Y 0 0 0 0	x-B0 y-B4 N0 (B0): S N1-3(B0): I N0-3 (B4): I	2. N0 esc. Y[0]
Después de evento 1	V 1 0 0 0 Y 1 0 0 0	x-B0 y-B4 N0 (B0): S N1-3(B0): I N0 (B4): M N1-3(B4): I	3. N1 lee X[1]
	V 1 1 0 0 Y 1 0 0 0	x-B0 y-B4 N0-1 (B0): S N2-3(B0): I N0 (B4): M N1-3(B4): I	4. N2 lee X[2]
Después de evento 2	V 1 1 1 0 Y 1 0 0 0	x-B0 y-B4 N0-2 (B0): S N3(B0): I N0 (B4): M N1-3(B4): I	5. N2 esc. Y[2]
	V 1 1 1 1 0 Y 0 0 1 0	x-B0 y-B4 N0-2 (B0): S N3(B0): I N2 (B4): M N0,1,3(B4): I	6. N1 esc. Y[1]
Después de evento 3	V 1 1 1 1 0 Y 0 0 1 0	x-B0 y-B4 N0-2 (B0): S N3(B0): I N1 (B4): M N0,2,3(B4): I	
	V 1 1 1 1 0 Y 0 1 0 0	x-B0 y-B4 N0-2 (B0): S N3(B0): I	

Notas sobre la notación usada en la tabla:

- ✓ N0(B0):S significa que el bloque 0 (B0) está en estado Compartido (S) en la caché del nodo 0 (N0).
- ✓ N1-3(B0):I significa que el bloque 0 (B0) está en estado Inválido (I) en las cachés de los nodos 1, 2 y 3.

problema 6. Se va a ejecutar en paralelo el siguiente código (initialmente x e y son 0):

F1	F2
x=1;	y=1;
x=2;	y=2;
print y ;	print x ;

Indicar los resultados que se pueden imprimir si (considerar que el compilador no altera el código):

(a) Se ejecutan F1 y F2 en un multiprocesador con consistencia secuencial.

(b) Se ejecutan F1 y F2 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 posibles resultados.

Solución

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

F1	F2
(1.1) x=1;	(2.1) y=1;
(1.2) x=2;	(2.2) y=2;
(1.3) print y ;	(2.3) print x ;

(a) Si F1 es el primero que imprime puede imprimir 0, 1 o 2, pero F2 podrá imprimir sólo 2. Esto es así porque se mantiene orden secuencial (los accesos a memoria del código que ejecuta un procesador se ven en el orden en el que están en dicho código) y, por tanto, cuando F1 lee y (instrucción (1.3) en el código), ya ha asignado a x un 2 (1.2) porque esta escritura está antes en el código que la lectura de y.

De igual forma, si F2 es el primero que imprime podrá imprimir 0, 1 o 2, pero P1 sólo puede imprimir 2. Esto es así porque se mantiene orden secuencial y, por tanto, cuando P2 lee x ((2.3) en el código), ha escrito ya en y un 2 (2.2) porque esta escritura está antes en el código que la lectura de x.

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

F1	F2
0	2 (en este caso F1 imprime 0 y F2 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:

E1	E2
1	1 (en este caso E1 imprime 1 y E2 imprime 1)
0	1
1	0
0	0

Se pueden imprimir también estas combinaciones porque no se asegura que cuando un procesador ejecute la lectura de la variable que imprime print ((1.3) y (2.3) en los códigos) haya ejecutado las instrucciones anteriores que escriben en x (F1 en los puntos (1.1) y (1.2)) o en y (F2 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. F1 puede leer y (1.3) antes de escribir en x 2 (1.2) o incluso antes de escribir en x 1 (1.1), por lo que F2 podría imprimir 0, 1 o 2, como F1, cuando F1 imprime primero. Igualmente, F2 puede leer x (2.3) antes de escribir en y 2 (2.2) o antes de escribir en y 1 (2.1), por lo que F1 podría imprimir 0, 1 o 2 como F2 cuando éste imprime primero. Todas las combinaciones son posibles.

Problema 7. Se va a ejecutar en paralelo el siguiente código (inicialmente x e y son 0):

F1	F2
x=30;	while (flag==0) {};
y=40;	r1=x;
flag=1;	r2=y;

Indicar qué datos puede obtener F2 en r1 y r2 si (considere que el compilador no altera el orden de los accesos a memoria del código):

- (a) Se ejecutan F1 y F2 en un multiprocesador con consistencia secuencial.
- (b) Se ejecutan en un multiprocesador con consistencia de liberación. Razoné su respuesta.

Solución

- (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 flujo:

- 1) En F1, 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 F1.

2) Hasta que F2 no encuentra en flag un 1 no almacena en r_1 el contenido de x ni en r_2 el contenido de y . Al mantenerse el orden $R \rightarrow R$, en F2 no se adelanta la lectura de x ni la lectura de y a la lectura de flag en la última iteración del bucle.

Por tanto, como se garantiza $W \rightarrow W$ y además se garantiza $R \rightarrow R$, si F2 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

r_1	r_2
0	0
0	40
30	0
30	40

Razonamiento:

1) Al no garantizarse el orden entre accesos de escritura ($W \rightarrow W$), F1 podría escribir en flag un 1 antes de escribir 30 en x y/o 40 en y (obsérvese que la escritura $y=40$ podría también adelantar a $x=30$). Por lo que F2 podría leer en flag un 1 y, por tanto, salir del bucle, antes de que en x y/o en y se pudiera ver los valores que escribe F1 en estas variables (x y/o y podrían ser 0).

2) Al no garantizarse el orden entre accesos de lectura ($R \rightarrow R$), si se permite ejecutar lecturas cuya ejecución depende de una condición de salto antes de verificar que se cumple la condición, en F2 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 F2 podría entonces leer los valores de x e y que tienen en su caché antes de que F1 los modifique y modifique flag. En estas circunstancias podría obtenerse en r_1 o en r_2 o en ambos un 0.



Problema 8. 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 tu respuesta. (b) ¿Cómo se debería implementar la función de liberación de un cerrojo simple si se usan procesadores Itanium? Razona tu respuesta.

Solución

(a) La función de liberación la utiliza un flujo después de acceder a las variables compartidas para permitir que otros flujos puedan acceder a estas variables. Los procesadores de la línea x86 sólo relajan $W \rightarrow R$. Teniendo esto en cuenta se podría implementar con la escritura “`mov k, 0`” la función de liberación puesto que en el multiprocesador no se permite que una escritura adelante a una lectura o escritura anterior y el acceso a variables compartidas está antes que el código de liberación; por tanto, la liberación

del cerrojo (escritura) no va a realizarse antes de haber terminado los accesos (lectura y escritura) a las variables compartidas.

(b) El procesador Itanium implementa un modelo de ordenación que relaja todos los órdenes. No obstante, proporciona 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 escritura en memoria con liberación (st.rel)

Problema 9. Se ha ejecutado el siguiente código en un multiprocesador con un modelo de consistencia que no garantiza W->R y W->W (garantiza el resto de órdenes):

```
(1) sump = 0;  
(2) for (i=ithread ; i<8 ; i=i+nthread) {  
    sump = sump + a[i];  
(3) }  
(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 y que la instrucción atómica de lectura-modificación-escritura no permite que lecturas posteriores adelanten a la escritura de la instrucción):

- (a) Indicar qué se puede obtener en sum si se suma la lista $a=\{1,2,3,4,5,6,7,8\}$. Se supone que 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 o flujo de instrucciones en el grupo (0,1,2) que está colaborando en la ejecución. 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? Justificar respuesta.

Solución

- (a) No hay problemas con el código de sincronización de adquisición porque $\text{Fetch_}\&_\text{Or}(k,1)()$ garantiza que la variable compartida sum se lee una vez que la escritura de 1 que cierra el cerrojo se ha realizado. Esto impide que el flujo que va a adquirir (cerrar) el cerrojo lea sum antes de que el flujo que tiene el cerrojo escriba en sum.

Al no garantizarse el orden W->W, la liberación del cerrojo, es decir la escritura de 0 en k, puede adelantar a los accesos a la variable compartida y, en particular, a la escritura de sum. Si esto ocurre, más de un flujo podría leer el mismo valor de sum, acumular a ese valor su variable local sump y escribir el resultado. En la variable sum acaba acumulado entonces, tras la escritura de los flujos que han leído lo mismo de sum, sólo el contenido de sump del último que ha escrito.

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

Lectura de sum	Resultado	Comentario
Todos ven distinto valor	$12+15+9=36$	Si no hay problemas debido a que la escritura de liberación adelante a los accesos a <code>sum</code> anteriores
Todos leen 0 de <code>sum</code>	12	Si los tres flujos de control llegan a leer el valor 0 inicial de <code>sum</code> y el último que escribe en <code>sum</code> tras actualizar su valor es el thread 0
	15	Si los tres flujos de control llegan a leer el valor 0 inicial de <code>sum</code> y el último que escribe en <code>sum</code> es el thread 1
	9	Si los tres flujos de control llegan a leer el valor 0 inicial de <code>sum</code> y el último que escribe en <code>sum</code> es el thread 2
Si dos leen el mismo valor en <code>sum</code> , y el otro un valor distinto	$12+15=27$ ó $12+9=21$	<ul style="list-style-type: none"> - Si el flujo de control 0 ha logrado acumular primero sin problemas y el 1 y el 2 leen el valor que tiene <code>sum</code> 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 (en <code>sump</code>) y escriben el resultado de la acumulación en <code>sum</code>. El resultado será 27 si el último que escribe es 1 y 21 si el último que escribe es 2. - 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 $15+9=24$	<ul style="list-style-type: none"> - Si el flujo de control 1 ha logrado acumular primero sin problemas y el 0 y el 2 acceden al valor que tiene <code>sum</code> justo tras la acumulación de 1. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 15, le acumulan cada uno su <code>sump</code> y escriben el resultado de la acumulación en <code>sum</code>. El resultado será 27 si el último que escribe es 0 y 24 si el último que escribe es 2. - 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 $9+15=24$	<ul style="list-style-type: none"> - Si el flujo de control 2 ha logrado acumular primero sin problemas y el 0 y el 1 acceden al valor que tiene <code>sum</code> justo tras la acumulación de 2. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 9, le acumulan cada uno su <code>sump</code> y escriben el resultado de la acumulación en <code>sum</code>. El resultado será 21 si el último que escribe es 0 y 24 si el último que escribe es 1. - 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.

E.T.S.I.I.T.
UPC

(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 `sum` en exclusión mutua (secuencial) por parte de los tres flujos. Cualquier orden es posible. 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 `sum` por todos los flujos de control: $12+15+9=36$.

Problema 10. ¿Qué ocurre si en el segundo código para implementar barreras visto en este Capítulo si se elimina 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

```
Barrera(id, num_flujos)
{
    band_local = !band_local
    (1)   lock(k);
    (2)   ++bar[id].cont; //Sec. Crítica
    (3)   unlock(k);
    (4)   if (bar[id].cont == num_flujos) {
    (5)       bar[id].cont=0;
    (6)       bar[id].band=band_local;
    (7)   }
    (8)   else while (bar[id].band != band_local) {};
}
```

Pueden surgir problemas pudiendo incluso no funcionar bien como barrera.

Si se usa el contador global en el `if` (línea de código (5)) que comprueba si contador es ya igual al número de flujos `num_flujos` que se sincronizan con la barrera, un flujo puede encontrar cuando llegue al `if` que el contador es igual a `num_flujos` sin ser el último que ha incrementado el contador. Esto puede provocar el siguiente problema:

Además del flujo que ha hecho el contador igual a `num_flujos`, 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 ((6) y (7)). 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 transferencia por cada variable a modificar sería suficiente. Estas transferencias adicionales simplemente devalúan prestaciones, nada más. El problema grave aparece si la barrera se vuelve a reutilizar por los mismos flujos y el SO suspende a uno de los flujos que encuentran contador igual a `num_flujos` antes de escribir un 0 en el contador global (6). Puede ocurrir entonces que, mientras este flujo está bloqueado, el resto de flujos salgan de la barrera y vuelvan a reutilizarla. Conforme los flujos van ejecutando de nuevo la barrera, incrementan el

contador global (3) y se quedan esperando a que éste llegue a ser igual a num_flujos (9). Pero nunca llegaría a alcanzar este valor porque cuando el flujo suspendido (antes de (a6)) vuelve a ejecutarse pondrá a 0 el contador global (ejecutará (6)) perdiéndose la cuenta del número de flujos que están ya esperando en la barrera.

Problema 11. Simplifique el segundo código para barreras visto en clase suponiendo que se ha implementado para una arquitectura que dispone de instrucciones `Fetch&Add()`.

Solución

A continuación se pueden ver los cambios realizados:

```
barrera(id, num_flujos) {
    bandera_local = !(bandera_local) //se complementa bandera local
    cont_local = fecht&Add(bar[id].cont,1); //Se incrementa contador
    if (cont_local == num_flujos-1) { //Si han llegado todos los flujos ...
        bar[id].cont = 0;           //... poner contador de flujos a 0 y ...
        bar[id].bandera = bandera_local; //... cambiar bandera para liberar ...
                                       //... flujos en espera
    }
} else while (bar[id].bandera != bandera_local) {} //Espera si no ...
                                                       //... han llegado todos los flujos
}
```

Se decrementa uno a num_flujos en el if porque `Fetch&Add()` devuelve el valor antes de la modificación, no después de la modificación. Por tanto, cuando llega el último flujo a la barrera devuelve num_flujos-1.

Problema 12. Se quiere parallelizar el siguiente bucle 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
}
```

Considerar que: (1) las iteraciones del bucle son independientes, (2) el único orden no garantizado por el sistema de memoria es W->R, (3) las instrucciones atómicas garantizan que escriben antes de que se puedan realizar lecturas posteriores y (4) el compilador no altera el código.

Teniendo esto en cuenta:

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

(b) Parallelizar el bucle en un multiprocesador que además tiene la primitiva `Fetch&Add()`.

Solución

$W \rightarrow R$.

(a) Usando `Fetch&Or()`:

Bucle secuencial original

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

Asignación Dinámica ($i=0$)

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

Se supone que el compilador no cambia de sitio $k=0$.

(b) Usando `Fetch&Add`:

Bucle secuencial original

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

Asignación Dinámica ($i=0$)

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

Problema 12. 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 su escritura antes que se ejecuten las lecturas posteriores):

```
Barrera(id, num_flujos)  
{  
    band_local= !(band_local)  
    while (Fetch&Or(k,1)==1) {};  
    cont_local = ++bar[id].cont;  
    k=0;  
    if (cont_local == num_flujos) {
```

```

        bar[id].cont=0;
        bar[id].band=band_local;
    }
    else while (bar[id].band != band_local) {};
}

```

Contestar razonadamente 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? ¿Por qué?
- (b) Funciona bien este código como barrera en un multiprocesador con modelo de consistencia de memoria de ordenación débil? ¿Por qué?

Solución

(R,W) atómica
R seguida de W (RAW)
W

 Barrera(id, num_flujos)
 {
 band_local = !(band_local)
 while (fetch_&_or(k,1)==1) {};
 cont_local = ++bar[id].cont; //sc
 k=0;
 if (cont_local == num_flujos) {
 bar[id].cont=0;
 bar[id].band=band_local;
 }
 else while (bar[id].band != band_local)
 }

Habrá problemas si dos o más flujos 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 flujo podría encontrar el cerrojo abierto estando otro flujo aún en la sección crítica (línea de código (3)), o
 2. El acceso a la variable compartida (lectura) adelanta a la operación atómica de escritura y lectura `Fetch&Or()`, porque el flujo que la adelanta accedería a la variable compartida (`bar[id].cont`) pudiendo estar otro flujo 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 (4) no puede adelantar a los accesos anteriores a la variable compartida `bar[id].cont` (3), puesto que la arquitectura no admite que una escritura pueda adelantar a accesos a memoria anteriores en el orden del programa. Por tanto, un flujo 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 (4) puede adelantar a los accesos anteriores a la variable compartida $\text{bar}[\text{id}].\text{cont}$ (3), puesto que la arquitectura permite que una escritura pueda adelantar a lecturas o escrituras anteriores en el orden del programa. Por tanto, un flujo puede asignar un 0 a k cuando aún no ha escrito en la variable compartida y , por tanto, otro podría entrar en la sección crítica por encontrarse el cerrojo abierto y podría leer el mismo valor de $\text{bar}[\text{id}].\text{cont}$ que el flujo que le abrió el cerrojo. Si esto ocurre, este contador nunca llegaría a num_flujos y todos los procesos se quedarían esperando en el bucle (9).

Problema 14. 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. El código lo ejecutan n threads en paralelo. Respecto a las variables, i es un identificador del flujo dentro del grupo de n threads; 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
...

```

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

- (a) Se ha ejecutado este código usando varios flujos y se ha visto que, aunque N y el vector x no varían, no siempre se imprime lo mismo. ¿Por qué ocurre esto?
- (b) Añadir lo mínimo necesario para solucionar el problema teniendo en cuenta que sólo se dispone para implementar sincronización de Test&Set (). Indique qué variables son ahora compartidas y cuáles locales.
- (c) Escribir el programa suponiendo que el multiprocesador además tiene instrucciones de sincronización Fetch&Add (). Indicar qué variables son compartidas y cuáles locales.

- (d) Escribir el programa ahora suponiendo que el multiprocesador sólo tiene instrucciones de sincronización `Compare&Swap()`. Indicar qué variables son compartidas y cuáles locales.

NOTA: En todos los apartados se puede añadir o quitar variables si se estima conveniente y se pide la implementación con mejores prestaciones

Solución

- (a) Hay varios 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 flujos (línea de código (5)). Esto permite que puedan intentar varios flujos a la vez acumular el resultado parcial que han calculado (carrera). Por ejemplo, varias pueden leer el mismo valor de `med`, acumular a ese valor su variable local `med1` y escribir el resultado en `med`. El resultado acumulado en `med` será entonces el valor que han leído todos los flujos más el contenido de `med1` de sólo una de ellos, en particular, del último que ha escrito en `med`. Por lo que no se acumularía lo calculado por todos ellos.
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 flujos. Esto puede llevar a restar varias veces a `vari` el resultado de elevar al cuadrado `med`. Para evitar este problema se puede introducir la línea (6) dentro del `if` que acompaña al `printf` para que el flujo 0 sea el único que modifique esta variable compartida.
3. El flujo 0 obtiene el valor definitivo de `vari` a partir de `med` y `vari` (6) e imprime (7) sin esperar a que todos los flujos acumulen en las variables compartidas `med` y `vari` los resultados parciales que han obtenido en el bucle en las variables locales `med1` y `varil`. Esto supone que, cuando imprime, pueden haber intentado acumular su resultado parcial el flujo 0 y una combinación del resto de flujos 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 dependiendo de cuántos flujos han realizado la acumulación antes de que el flujo 0 imprima el resultado.

- (b) Se accederá en exclusión mutua a `med` y `vari` (punto 1 de (a)) implementando un cerrojo simple con una variable compartida `k1`, se tiene que añadir una barrera antes de que imprima el flujo 0 (punto 3) y se introduce (6) dentro del `if` (punto 2). La barrera se debe implementar también usando un cerrojo simple (`k2`). El código resultante sería el siguiente:

```

(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) Punto 1
        med = med + medl; vari = vari + varil;
                                //unlock(k1) Punto 1
        k1=0;
        bandera_local= !(bandera_local)
        while (Test&Set(k2)) {}; //lock(k2)
        bar[id].cont+=1; cont_local = bar[id].cont;
                                //unlock(k2)
        k2=0;
        if (cont_local == num_flujos) {
            if (bar[id].cont==0;
                bar[id].cont=0;
                bar[id].bandera= bandera_local;
            }
            else while (bar[id].bandera!= bandera_local) {}; //Punt. 3
        }
        if (ithread==0) { vari= vari - med*med; //Punto 2
                        printf("varianza = %f", vari);}
    
```

bandera_local, cont_local, ithread, i, medl y varil son variables locales del flujo; k1, k2, bar[], med, vari, x[] y N son variables compartidas; inicialmente k1, k2, med, vari, medl y varil son 0.

(c) Con Fetch&Add() no es necesario usar variables compartidas extras como cerrojos para el acceso en exclusión mutua:

```

(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); //Punto 1
        bandera_local= !(bandera_local) //Inicio Punto 3
        cont_local = Fetch&Add(bar[id].cont,1);
        if (cont_local==num_flujos-1) {
            bar[id].cont=0;
            bar[id].bandera= bandera_local; }
            else while (bar[id].bandera!= bandera_local) {}; //Punt. 3
    
```

- (6) if (ithread==0) { vari= vari - med*med; //Punto 2
 printf("varianza = %f", vari);}

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

(d) Con Compare&Swap() no es necesario usar variables compartidas extras como cerrojos para el acceso en exclusión mutua:

```

(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;
(5)   do                                //Inicio Punto 1
(5)     a = med;
(5)     b = a + medl; //a y b son variables locales
(5)     Compare&Swap(a,b,med);
(5)   while (a!=b);
(5)   do
(5)     a = vari;
(5)     b = a + varil;
(5)     Compare&Swap(a,b,vari);
(5)   while (a!=b);                      //Punto 1
bandera_local= !(bandera_local)
do                                //Inicio Punto 3
  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_flujos-1) {
  bar[id].cont=0;
  bar[id].bandera= bandera_local;
} else while (bar[id].bandera!= bandera_local) {}; //Punt. 3
(6) if (ithread==0) { vari= vari - med*med;           //Punto 2
(7)           printf("varianza = %f", vari);}

```

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

Problema 15. 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
#ifndef NR_CPUS < 256
...
static __always_inline void __ticket_spin_lock(raw_spinlock_t *lock)
{
  short inc = 0x0100;
}

```

```

-     asm volatile (
-         "lock xaddw %w0, %l\n\t" /*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 zf \n\t" /*f: forward */
-         "rep ; nop \n\t" /*retardo, es equivalente a pause*/
-         "movb %l, %b0 \n\t" /*don't need lfence here, because loads are in-order */
-         /* don't need lfence here, because loads are in-order */
-         "\n\t" /*b: backward */
-         "jmp 1b \n"
-         "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:

- Utiliza una implementación de cerrojo con etiquetas. ¿Cuál es el contador de adquisición y cuál es el contador de liberación?
- Describir qué hace xaddw %w0, %l ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?
- Describir qué hace cmpb %h0, %b0 ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?
- ¿Por qué se usa el prefijo lock delante de la instrucción xaddw?

NOTA: Se pueden ver detalles de la interfaz entre C/C++ y ensamblador en el manual de gcc (<http://gcc.gnu.org/onlinedocs/>)

Solución

- lock->slock contiene el contador de liberación en los bits de 0 a 7 (lock->slock[7...0]) y el de adquisición en los bits de 8 a 15 (lock->slock[15...8]).
- xaddw %w0, %l almacena los 16 bits (sufijo w) menos significativos de lock->slock (%l) en los 16 bits menos significativos del registro al que se ha asigna inc (%0) 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]`).

(c) `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)`

(d) 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.

7. Cuestiones

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

Respuesta

Un núcleo con multithread simultánea es un núcleo superescalar que puede emitir a unidades funcionales instrucciones de múltiples threads o flujos de instrucciones 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), lo que hace es multiplexar en el tiempo el uso de las etapas del cauce por parte de 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.

Respuesta

En un multithread temporal de grano fino (FGMT- *Fine Grain MultiThreading*) el hardware puede conmutar de thread cada ciclo de reloj (con una penalización de 0 ciclos) (1) por turno rotatorio o (2) por un evento (dependencia funcional, fallo en caché de nivel 1, operación con CPI de varios ciclos, salto no predecible) combinado con

algunas técnicas de planificación (p. ej. pasar a ejecutar el thread que lleva menos tiempo sin ejecutarse).

En un multithread temporal de grano grueso (CGMT- *Coarse Grain MultiThreading*) se conmuta de thread tras varios ciclos de reloj (con una penalización que puede ser distinta de 0) (1) trascurrido un nº de ciclos prestablecido o (2) por un evento estático (ejecución de una instrucción añadida al repertorio o ya incorporada, en este último caso podrían ser instrucciones de carga/almacenamiento o de salto) o dinámico (como el fallo de la caché de último nivel o una interrupción).

Cuestión 3. Se tiene un multiprocesador con protocolo MESI de espionaje. Si un controlador de caché observa en el bus un paquete de petición de lectura exclusiva de un bloque que tiene en estado S, debe (indicar cuál sería la respuesta correcta y razonar 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.

Respuesta

- 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. Se tiene 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? Razonar respuesta.

Respuesta

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 enviado el paquete va a escribir en su caché en la copia que va a recibir, por tanto, la copia que está en estado M ya no estará actualizada, será inválida.

Answer thread

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 caché en estado I, debe (indique cuál sería la respuesta correcta y razoné por qué es la respuesta correcta):

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

Respuesta

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 caché del nodo no tiene copia válida del bloque (debido al estado Inválido), el controlador de caché 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 caché del nodo tiene que pedir acceso exclusivo para que las copias del bloque en otras cachés 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 en lugar de acceder a copias no actualizadas del bloque que estén en sus cachés.
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 y, por tanto, deberá estar en estado Modificado para que el controlador de caché sepa que debe responder con el bloque si se recibe alguna petición que incluya lectura del bloque y que debe escribirlo en memoria si se reemplaza el bloque en la caché.

Cuestión 6. ¿Cuál de los siguientes modelos de consistencia permite mejores tiempos de ejecución?. Justificar 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

Respuesta

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 ($W \rightarrow R$, $W \rightarrow W$, $R \rightarrow W, R$) 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 modelo de ordenación débil ofrece instrucciones máquina que permiten que, si S es una operación de sincronización, se garanticen los siguientes órdenes (W y L son operaciones de escritura y lectura, respectivamente):

- $S \rightarrow WR$ (hasta que no termina la operación de sincronización no puede empezar ningún acceso a memoria posterior)
- $WR \rightarrow S$ (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, independientemente de si se trata de una sincronización de adquisición o liberación. Por tanto, se garantiza el siguiente orden entre operaciones de acceso a memoria y operaciones de sincronización de adquisición SA y de liberación SL :

$WR \rightarrow SA \rightarrow WR \rightarrow SL \rightarrow WR$ (ver Figura 9)

Teniendo esto en cuenta se deduce que no hay ningún tipo de paralelismo entre operaciones de acceso a memoria antes y después de operaciones de sincronización.

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

- $SA \rightarrow WR$ (hasta que no termina la operación de sincronización de adquisición no puede empezar ningún acceso a memoria posterior)
- $WR \rightarrow SL$ (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 (ver Figura 10). 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. Indicar qué función no se corresponde con la serie (justificar respuesta):

- a) lock()
- b) Fetch&Or()
- c) Compare&Swap()
- d) Test&Set()

Respuesta

Hay una función software de cerrojos, lock(), y tres instrucciones máquina utilizadas para mejorar prestaciones en la sincronización de flujos de instrucciones: Fetch&Or(), Compare&Swap() y Test&Set().

Luego lock() no se corresponde con la serie.

8. Bibliografía

- S.V. Adve, K. Gharachorloo, 1996. "Shared Memory Consistency Models: A Tutorial", IEEE Computer, vol. 19, nº 12, diciembre.
- ARM, 2015. "ARM Architecture Reference Manual. ARMv8."
- Julio Ortega Lopera, Mancia Anguita López y Alberto Prieto, 2005. "Arquitectura de Computadores". Thomson.