2º curso / 2º cuatr.

Grado en Ing. Informática

Arquitectura de Computadores. Algunos 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 **N2) Inválido** **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 2.-MP (el controlador de memoria de MP), al observar	**N1) Exclusivo** **N2) Inválido** **N3) Inválido**

		PtLec(k) en el bus, genera la respuesta con el bloque (RpBloque(k)). 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 **N2) Inválido** **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 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)). 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** **N2) Compartido** **N3) Inválido**
N1) Compartido **N2) Compartido** **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. 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** **N2) Inválido** **N3) Inválido**
N1) Modificado **N2) Inválido** **N3) Inválido**	P2 escribe en k	1.- N2 genera petición de lectura con acceso exclusivo de k (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 del bloque k. 3.- N2 recoge RpBloque(k), introduce k en su cache, lo modifica y lo pone en estado modificado	**N1) Inválido** **N2) Modificado** **N3) Inválido**
N1) Inválido **N2) Modificado** **N3) Inválido**	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 cache, lo modifica y lo pone en estado modificado	**N1) Inválido** **N2) Inválido** **N3) Modificado**

◀◀◀◀◀◀

Ejercicio 2. .

Ejercicio 3. .

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.

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 ;`

(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`".

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	
0	2	
1	0	
2	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.

◀

Ejercicio 6. .

◀

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

(a) 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. - 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. - 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. - 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. .

◀

Ejercicio 9. .

◀

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.

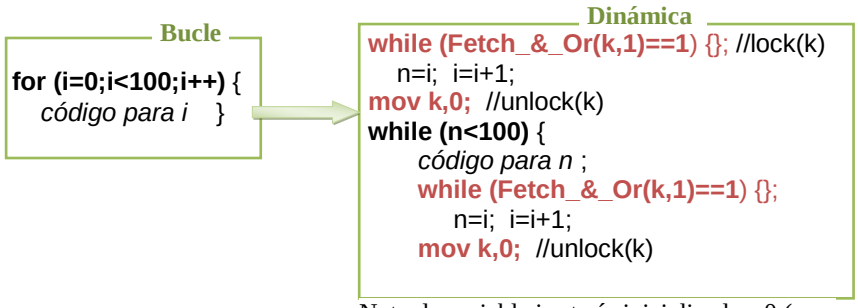
(a) Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva `Fetch&Or` para garantizar exclusión mutua.

(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.

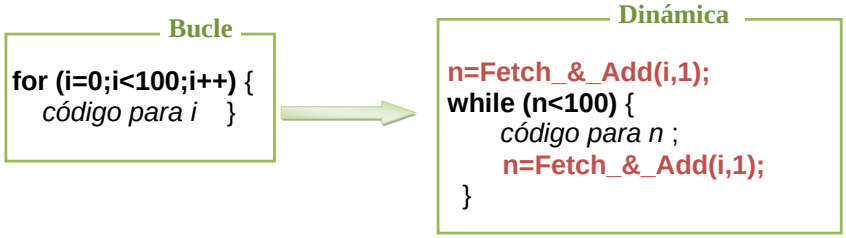
(a) Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva `Fetch&Or` para garantizar exclusión mutua.



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`.



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

Ejercicio 11.

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)
(7)  if (ithread==0) { vari= vari - med*med;
                       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) {};

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

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.