

### 3. Tema 3

#### 3.1. Arquitecturas TLP

##### 3.1.1. Clasificación de arquitecturas con TLP explícito y una instancia de SO

- Los **multiprocesadores** ejecutan varios threads en paralelo en un computador (cada thread en un core distinto).
- Los multiprocesadores en un chip (CMP) o **multicore** ejecutan varios threads en paralelo en un chip de procesamiento multicore (cada thread en un core distinto).
- Los **core multithread** son cores que modifican su arquitectura ILP para ejecutar threads concurrentemente (en paralelo)

##### 3.1.2. Multiprocesadores

Ejecutan varios threads en paralelo en un computador (cada thread en un core distinto).

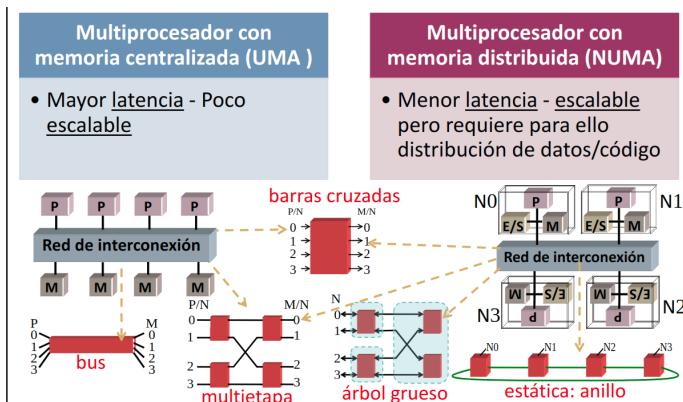


Figura 32: Tipos de multiprocesadores según sistema de memoria.

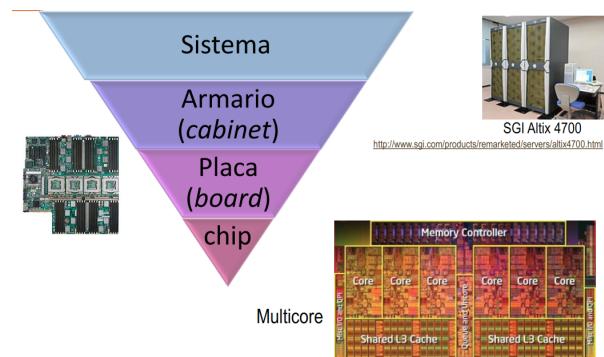


Figura 33: Tipos de multiprocesadores según empaquetamiento.

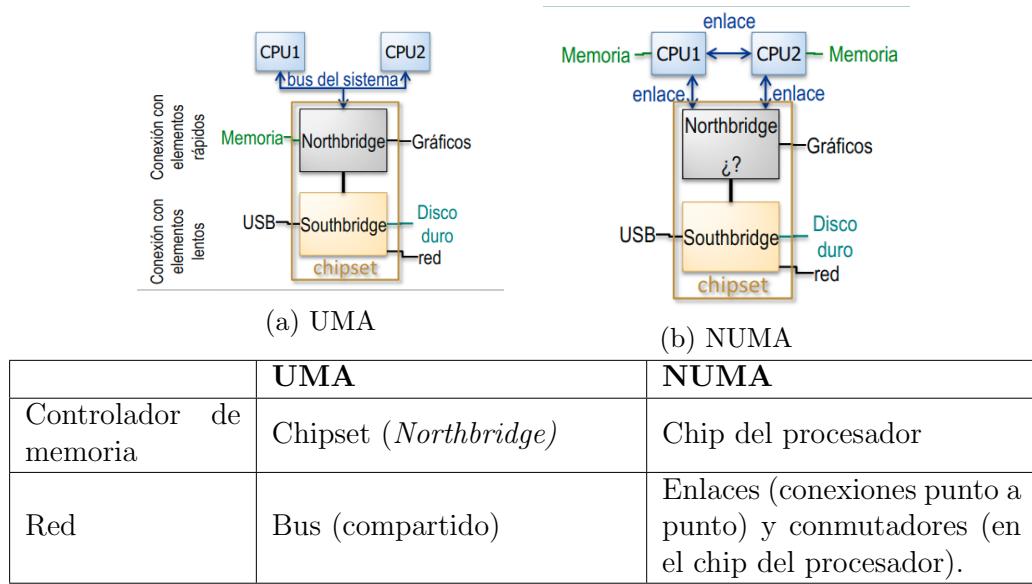


Figura 34: Diferencias entre UMA y NUMA.

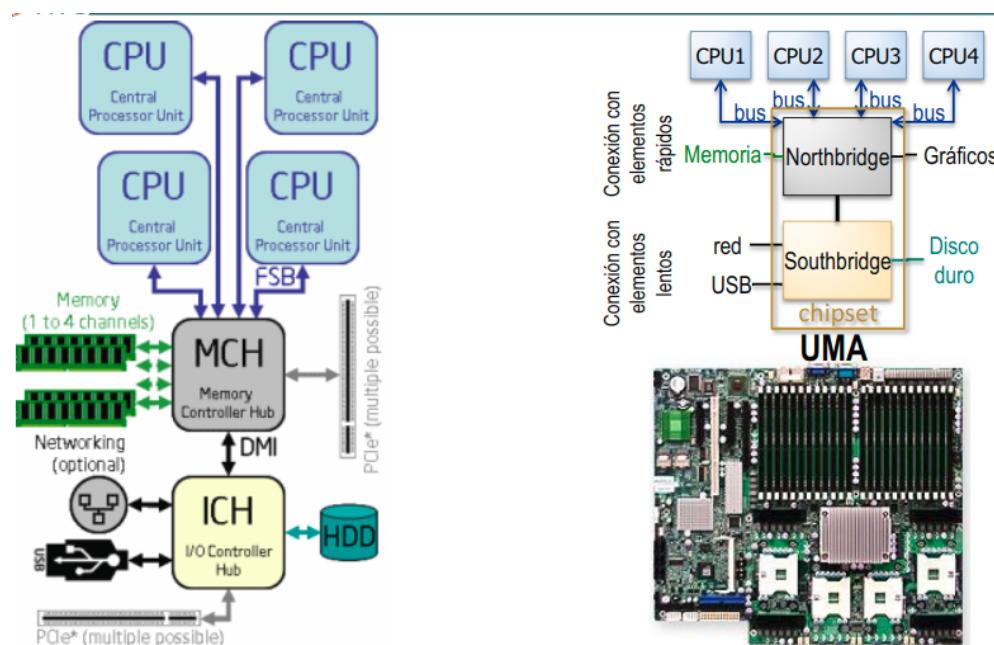


Figura 35: Placa base y CPU UMA.

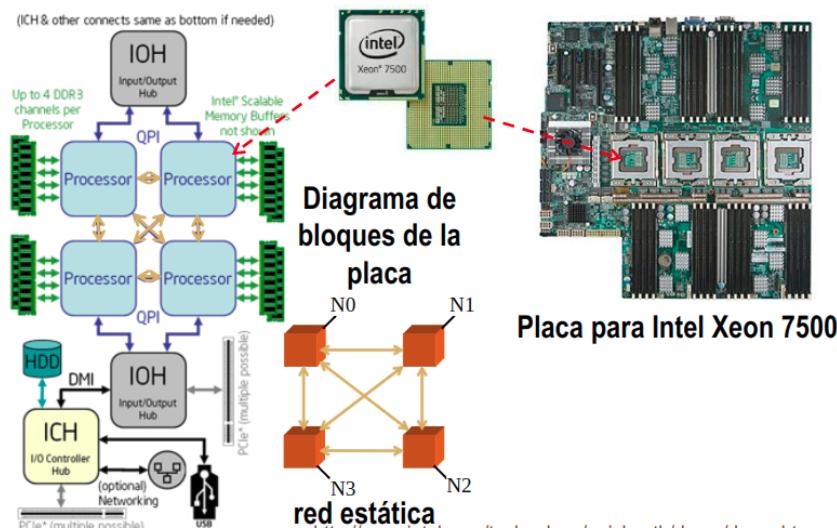


Figura 36: Placa base y CPU NUMA.

### 3.1.3. Multicores

Ejecutan varios threads en paralelo en un chip de procesamiento multicore (cada thread en un core distinto).

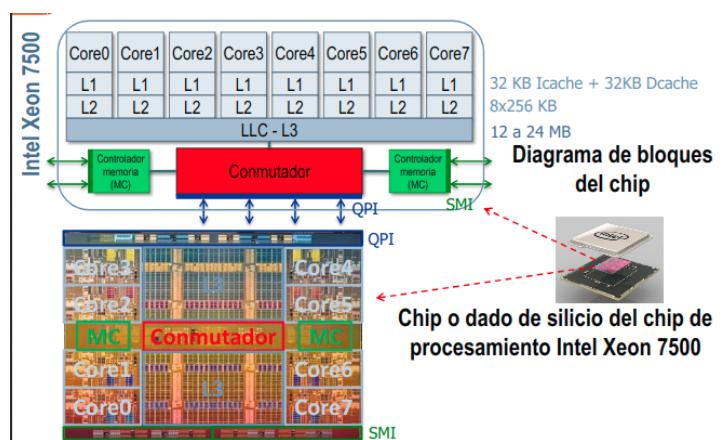


Figura 37: Ejemplo de multicore.

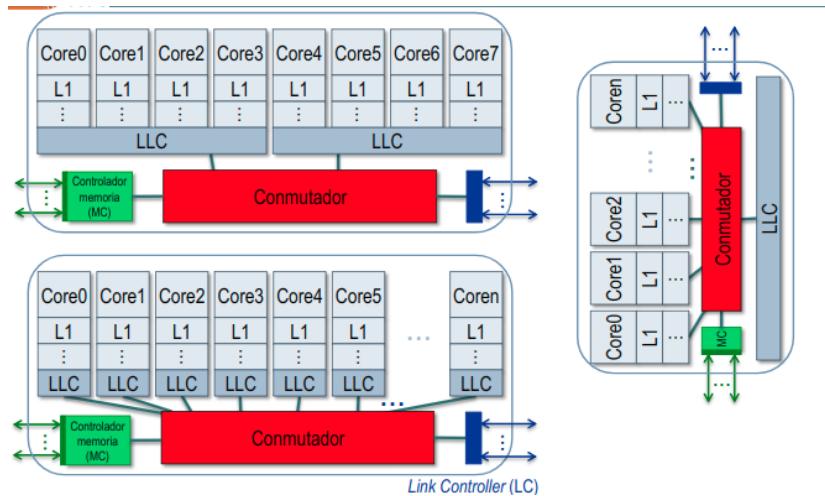


Figura 38: Posibles arquitecturas de multicore.

### 3.1.4. Cores multithread

Modifican su arquitectura ILP (segmentada, escalar o VLIW) para ejecutar threads concurrentemente o en paralelo.

Las operaciones tienen varias estapas:

1. Captación (Instruction Fetch).
2. Decodificación y emisión a unidades funcionales (Instruction Decode).
3. Ejecución (Execution) y acceso a memoria (Memory).
4. Almacenamiento de resultados (Write-Back).

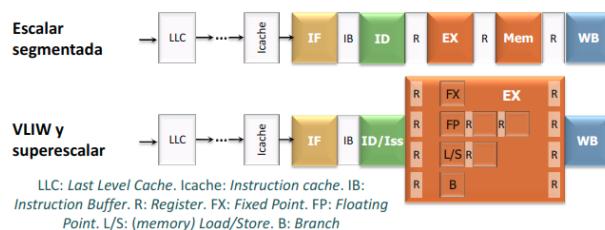


Figura 39: Tipos de arquitecturas ILP.

- Los procesadores **segmentados** ejecutan instrucciones concurrentemente segmentando el uso de sus componentes.
- Los procesadores VLIW (*Very Large Instruction Word*) y los superescalares ejecutan las instrucciones concurrentemente (segmentación) y en paralelo (tienen múltiples unidades funcionales).

- VLIW
  - Las instrucciones que se ejecutan en paralelo se captan juntas, formando la VLIW.

- El paralelismo (de instrucciones) es extraído mediante software.
- Superescalares. El paralelismo de instrucciones es extraído mediante hardware específico.

Los cores multithread se pueden clasificar en dos grandes grupos:

- *Temporal Multithreading (TMT)*.
  - Ejecutan varios threads concurrentemente en el mismo core.
  - La conmutación entre threads es gestionada por el hardware.
  - Emite instrucciones de un único thread en un ciclo.
- *Simultaneus Multithreading (SMT)*, multihilo simultáneo o *horizontal multithread*.
  - Ejecutan en un core superescalar varios threads en paralelo.
  - Pueden emitir instrucciones de varios threads en un ciclo.

A su vez, los cores TMT se pueden dividir en:

- *Fine-grain multithreading (FGMT)* o *interleaved multithreading*. La conmutación entre threads la decide el hardware en cada ciclo (coste 0). Utiliza algoritmos de planificación, como *Round-Robin*.
- **Coarse-grain mutithreading (CGMT)** o *blocked multithreading*. La conmutación entre threads la decide el hardware (coste 0) en varios ciclos. Pueden ser en intervalos de tiempo prefijados o bien por eventos de cierta latencia.

Por último, los cores con CGMT y conmutación por eventos se clasifican en:

- **Estática**. La conmutación puede ser explícita (instrucciones añadidas al repertorio) o implícita (usando instrucciones existentes). Su principal ventaja es que el coste del cambio de contexto es bajo (0/1 ciclo) y su inconveniente que se realizan cambios de contexto innecesarios.
- **Dinámica**. La conmutación se realiza por un fallo en la última caché, una interrupción, etc. Su principal ventaja es que reduce los cambios de contexto innecesarios pero aún así se añade sobrecarga al realizar cambios de contexto.

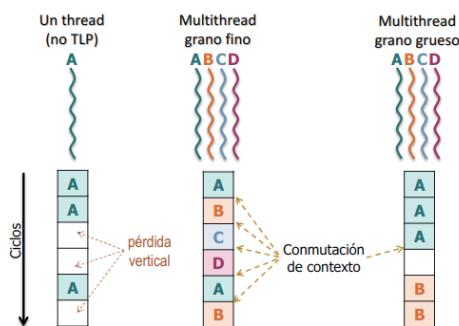


Figura 40: Alternativas en un core escalar segmentado (se emite una instrucción por ciclo).

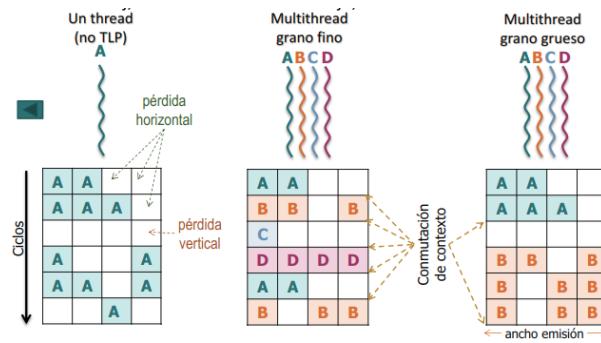


Figura 41: Alternativas en un core superescalar o VLIW (se emiten varias instrucciones por ciclo pertenecientes a un único thread).

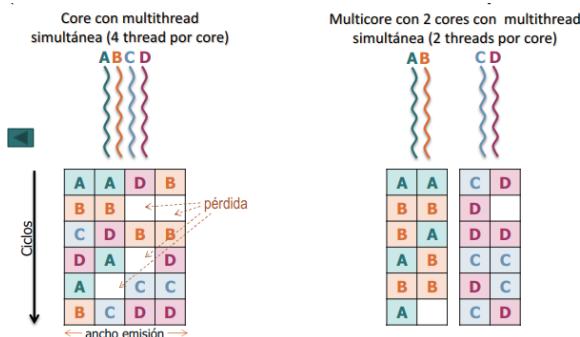


Figura 42: Alternativas en multicore (se emiten instrucciones de distintos threads en cada ciclo).

### 3.1.5. Hardware y arquitecturas TLP en un chip

Hardware	CGMT	FGMT	SMT	CMP
Registros	replicado (al menos PC)	replicado	replicado	replicado
Almacenamiento	multiplexado	multiplexado, compartido, repartido o replicado	compartido, repartido o replicado	replicado
Otro hardware de las etapas del cauce	multiplexado	Captación: repartida o compartida; Resto: multiplexadas	UF: compartidas; Resto: repartidas o compartidas	replicado
Etiquetas para distinguir el thread de una instr.	Sí	Sí	Sí	No
Hardware para comutar entre threads	Sí	Sí	No	No

Figura 43: Hardware y arquitecturas TLP en un chip.

ABIERTA CONVOCATORIA 2022

Work  
to  
change  
your  
future



### 3.2. Coherencia del sistema de memoria

#### 3.2.1. Sistema de memoria en multiprocesadores

El uso de jerarquía de memoria con el fin de acercar la velocidad de acceso a memoria a la del procesador hace posible que pueda haber varias copias de un mismo bloque en el sistema. Ésto puede ocurrir tanto en multiprocesadores como en uniprocesadores. El *sistema de memoria* incluye las cachés de todos los nodos, memoria principal, controladores, buffers (escritura, combinación de escrituras, etc.) así como el medio de comunicación de estos componentes (red de interconexión).

La comunicación de datos entre procesadores es realizada por el sistema de memoria. Para que sea coherente, la lectura de una dirección debe devolver lo último que se ha escrito (desde el punto de vista de todos los componentes del sistema).

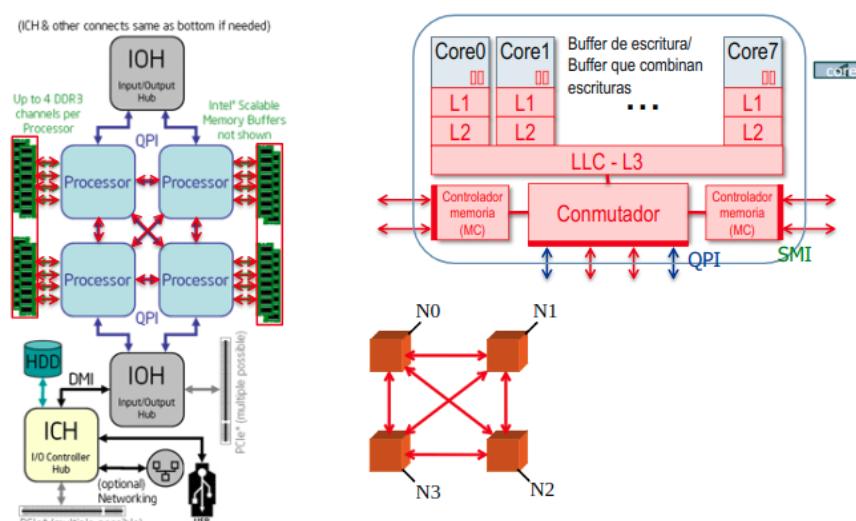


Figura 44: Sistema de memoria.

#### 3.2.2. Concepto de coherencia en el sistema de memoria: situaciones de incoherencia y requisitos para evitar problemas en estos casos

Si en el sistema de memoria las copias de una dirección no tienen el mismo contenido, tendremos una **incoherencia en el sistema de memoria**. En sistemas con una sola caché se puede dar incoherencia entre la caché y la memoria principal. En sistemas con múltiples cachés la incoherencia se puede dar además entre las distintas cachés.

Clases de estructuras de datos	Eventos que ponen de manifiesto faltas de coherencia	Tipos de Falta de coherencia
Datos modificables	E/S	Cache-MP
Datos modificables compartidos	Fallo de cache	Cache-MP
Datos modificables privados	Emigra thread/proceso → Fallo cache	Cache-MP
Datos modificables compartidos	Lectura de cache no actualizada	Cache-Cache

Figura 45: Tipos de estructuras de datos y faltas de coherencia.

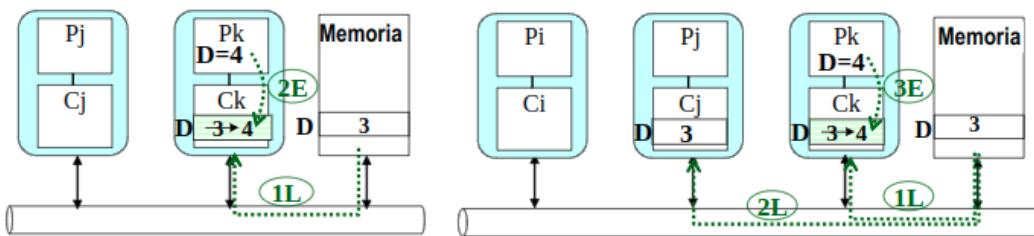


Figura 46: Ejemplo de incoherencia.

### Métodos de actualización de MP implementados en cachés.

- **Escritura inmediata** (*write-through*). Cada vez que un procesador escribe en su caché escribe también en MP. Por tanto, cada escritura supone el uso de la red. No evita las incoherencias entre cachés.
- **Posescritura** (*write-back*). La escritura se realiza cuando el bloque que contiene la dirección modificada se elimina de la caché para alojar un nuevo bloque. Con la posescritura se permiten incoherencias de caché y MP.

La falta de coherencia entre cachés y los problemas de incoherencias se pueden solventar mediante hardware específico de coherencia. Existen alternativas para abordar las incoherencias entre caché y MP si el sistema no tiene hardware específico de mantenimiento de coherencia. Por ejemplo, se puede evitar la falta de coherencia declarando las zonas de memoria implicadas en la comunicación como *no cacheables* o bien con escritura inmediata.

Como ya hemos visto, en sistemas con múltiples cachés puede haber faltas de coherencia entre cachés, tanto con escritura inmediata como con posescritura. Los **protocolos de coherencia de caché** resuelven este problema haciendo que cada escritura sea visible a todos los procesadores, propagando de forma fiable un nuevo valor escrito en una dirección.

### Alternativas para propagar una escritura en protocolos de coherencia de caché.

- **Escritura con actualización** (*write-update*). Cada vez que un procesador escribe en una dirección en su caché, se escribe el nuevo dato en las copias de esa dirección de las otras cachés.

- **Escritura con invalidación** (*write-invalidate*). Antes de que un procesador modifique una dirección en su caché se invalidan las copias del bloque de la dirección en otras cachés. Cuando otro procesador quiera leer la información tendrá un fallo de caché, por lo que obtendrá el dato actualizado.

Hay que tener en cuenta que puede ocurrir que la propagación de una escritura a todos los procesadores a través de la red de interconexión requiera un tiempo distinto para cada procesador. En ese caso, si se solaparan varias propagaciones, distintos procesadores podrían recibir y leer en distinto orden las escrituras, causando incoherencia.

### Requisitos del sistema de memoria para evitar incoherencias

- La escritura en una dirección debe hacerse visible en un tiempo finito a los otros procesadores.
- Las escrituras en una misma dirección deben verse en el mismo orden por todos los procesadores. Es decir, las operaciones de escritura en la misma dirección se deben realizar en serie (una detrás de otra).

Las comunicaciones uno a muchos para actualizar o invalidar las copias del bloque que se modifica se pueden implementar en un bus, ya que permite la difusión de forma natural. En buses se utilizan **protocolos de espionaje (snoopy)**, que aprovechan que todos los componentes conectados al bus pueden ver (espiar) su contenido. El controlador de caché espía los parquetes en el bus y actúa en consecuencia (por ejemplo, invalidando o actualizando la copia que tienen de un bloque).

En redes en las que la difusión es costosa de implementar o en las que se requiere una mayor escalabilidad se usan **esquemas basados en directorios**. En ellos se envían paquetes únicamente a las cachés implicadas para reducir el tráfico. La información sobre las cachés implicadas en cada momento se almacena en un directorio asociado a la memoria principal.

Este directorio centralizado (monolítico), al igual que el bus, supone un cuello de botella. Podemos obtener mejores prestaciones si lo distribuimos entre los módulos de memoria principal. Es decir, el subdirectorío de cada módulo mantiene información de los bloques que conteinen el módulo. Los diferentes subdirectorios procesan peticiones en paralelo.

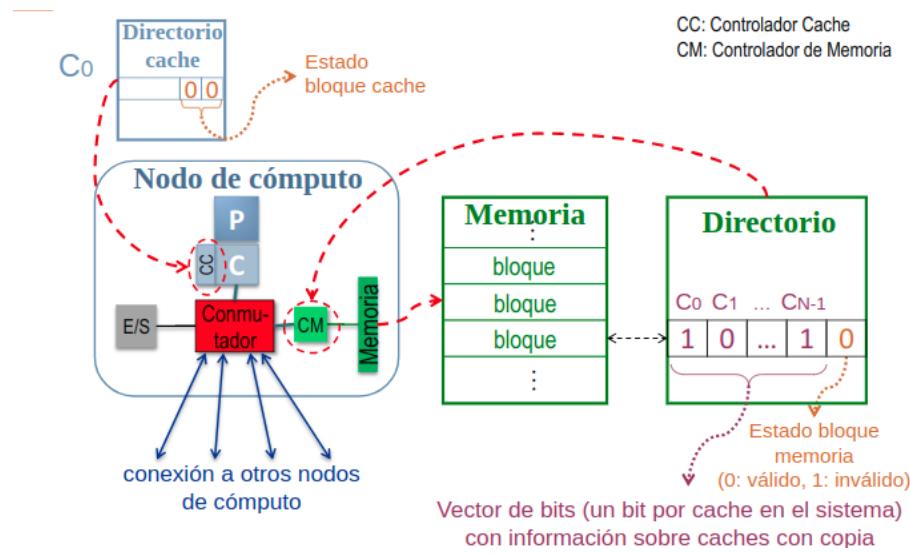


Figura 47: Directorio de memoria principal.



Figura 48: Alternativas para implementar el directorio.

El hecho de que las operaciones de acceso a memoria tengan que llegar al directorio para obtener información de un bloque (cachés implicadas) permite usar el directorio para establecer un orden entre accesos a una misma dirección (serializar) de forma que todos los procesadores vean este mismo orden. No obstante, en algunos sistemas se pueden necesitar paquetes de reconocimiento de actualizaciones o invalidaciones que confirmen que ha finalizado una escritura anterior e información de estado para el bloque en el directorio (si está pendiente de finalizar un acceso).

También se pueden encontrar esquemas de coherencia de caché organizados en **jerarquía** compuestos de protocolos de espionaje y directorios, según la red de cada nivel.

- Protocolos de **espionaje** (*snoopy*). Para buses y en general sistemas con difusión eficiente (pocos nodos o red que implementa difusión).
- Protocolos basados en **directorios**. Para redes sin difusión o escalables.
- Esquemas **jerárquicos**. Se usan para redes jerárquicas: jerarquía de buses, de redes escalables, redes escalables-buses, etc.

Figura 49: Protocolos para mantener coherencia en el sistema de memoria.



### 3.2.3. Protocolo MSI de espionaje

También se denomina protocolo de invalidación de tres estados. Utiliza posescritura y escritura con invalidación.

Un bloque puede tener tres estados:

- **Modificado (M).** Es la única copia válida del bloque en todo el sistema. La caché debe proporcionar el bloque si al espiar el bus observa que algún componente la solicita e invalidarlo si algún nodo solicita una copia exclusiva para su modificación.
- **Compartido (S).** Todas las copias del bloque están actualizadas. La caché debe invalidar su copia si al espiar el bus observa que algún nodo solicita una copia exclusiva del bloque para su invalidación.
- **Inválido (I).** El bloque se ha invalidado o bien no se encuentra físicamente en caché.

Por su parte, un bloque puede ser:

- **Válido.** Tiene una copia válida. Es decir, el controlador puede proporcionar el bloque en caso de solicitud.
- **Inválido.** La copia no es la última actualización, es decir, hay una caché que ha modificado el bloque (estado modificado).

Un nodo terminal puede generar las siguientes transferencias como consecuencia de acciones de lectura/escritura en la caché del procesador o bien como consecuencia de paquetes recibidos de otros nodos:

- Petición de lectura de un bloque (**PtLec**). Un procesador lee de una dirección que no se encuentra en su caché (fallo de caché). El controlador inicia la transacción poniéndole en el bus la dirección a la que desea acceder. El sistema de memoria proporcionará el bloque donde se encuentra la dirección solicitada.
- Petición de acceso exclusivo (**PtLecEx**). Se genera por la escritura de un procesador (PrEsc) en un bloque compartido o inválido. El controlador de caché genera el paquete que indicará la dirección en la que se desea escribir. El resto de cachés con copias válidas las invalidarán. También se invalidará el bloque en memoria si se encuentra en estado válido. El procesador puede solicitar una confirmación de este paquete.
- Petición de posescritura (**PtPEsc**). Se genera por el reemplazo de un bloque modificado debido a un fallo de caché (se sobreescribe el bloque con estado modificado). El procesador del nodo no conoce este hecho y no espera respuesta.
- Respuesta con bloque (**RpBloque**). La genera un nodo que observa en el bus una petición de un bloque que tiene en estado modificado solicitada por una petición de lectura (exclusiva o no).

EST. ACT.	EVENTO	ACCIÓN	SIGUIENTE
<b>Modificado (M)</b>	PrLec/PrEsc		Modificado
	PtLec	Genera paquete respuesta (RpBloque)	Compartido
	PtLecEx	Genera paquete respuesta (RpBloque) Invalida copia local	Inválido
	Reemplazo	Genera paquete posescritura (PtPEsc)	Inválido
<b>Compart. (S)</b>	PrLec		Compartido
	PrEsc <small>POSESCR</small>	Genera paquete PtLecEx (PtEx)	Modificado
	PtLec		Compartido
	PtLecEx <small>Inválido</small>	Invalida copia local	Inválido
<b>Inválido (I)</b>	PrLec	Genera paquete PtLec	Compartido
	PrEsc	Genera paquete PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

Figura 50: Descripción de MSI.

### 3.2.4. Protocolo MESI de espionaje

También denominado protocolo de invalidación de cuatro estados. También utiliza posescritura y escritura con invalidación.

En el protocolo MSI, si un procesador escribía en un bloque de su caché tenía que generar una transferencia que invalidase copias en otras cachés, aunque no hubiera copias del bloque en cachés de otros procesadores. Este hecho degrada las prestaciones de aplicaciones secuenciales ejecutadas en multiprocesadores, ya que no comparten variables entre procesos.

Para solventar este problema se añade un nuevo estado: *exclusivo*. Un bloque en estado exclusivo será válido sólo en la caché en la que se encuentra y en memoria. Si un bloque se encuentra en estado exclusivo en la caché de un nodo y el procesador del nodo escribe en el bloque, no se genera paquete para solicitar uso exclusivo e invalidar copias, ya que sabemos que ninguna otra caché tiene copia válida.

Los posibles estados de un bloque son:

- **Modificado (M)**. Es la única copia válida en todo el sistema.
- **Exclusivo (E)**. Es la única caché en el sistema con una copia válida del bloque, el resto tienen una copia no válida. La memoria tiene también una copia actualizada del bloque. La caché debe invalidar su copia si observa que algún otro nodo solicita una copia exclusiva del bloque para su modificación.
- **Compartido (C)**. El bloque es válido, también en memoria y en al menos alguna otra caché. La caché debe invalidar su copia si observa que algún otro nodo solicita una copia exclusiva del bloque para su modificación.
- **Inválido (I)**. El bloque no está físicamente en caché o bien está en estado no válido.

	PrLec/PrEsc		Modificado
Modificado (M)	PtLec	Genera RpBloque	Compartido
	PtLecEx	Genera RpBloque. Invalida copia local	Inválido
	Reemplazo	Genera PtPEsc	Inválido
Exclusivo (E)	PrLec		Exclusivo
	PrEsc		Modificado
	PtLec		Compartido
	PtLecEx	Invalida copia local	Inválido
Compartido (S)	PrLec/PtLec		Compartido
	PrEsc	Genera PtLecEx	Modificado
	PtLecEx	Invalida copia local	Inválido
Inválido (I)	PrLec (C=1)	Genera PtLec	Compartido
	PtLec (C=0)	Genera PtLec	Exclusivo
	PrEsc	Genera PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

Figura 51: Descripción de MESI. Quitando lo tachado se obtiene MSI.

### 3.2.5. Protocolo MSI basado en directorios con o sin difusión

**MSI con directorios sin difusión** Un bloque en caché puede tener tres estados distintos: **modificado (M)**, **compartido (C)** o **inválido (I)**.

Un bloque en MP puede ser **válido** o **inválido**.

Hay varios tipos de nodos: **solicitante (S)**, origen (O), modificado (M), propietario (P) y compartidor (C).

Si un nodo **S** realiza una petición a un nodo **O**, la petición puede ser:

- Lectura de bloque (**PtLec**).
- Lectura con acceso exclusivo (**PtLecEx**).
- Petición de acceso exclusivo sin lectura (**PtEx**).
- Posescritura (**PtPEsc**).

Los nodos **O** pueden reenviar a los nodos con copia (**P, M o C**) varias peticiones: invalidación (**RvInv**) o lectura (**RvLec** o **RvLecEx**).

También pueden recibir respuestas:

- Nodo **P** a **O**: respuesta con bloque (**RpBloque**) o respuesta con/sin bloque confirmando invalidación (**RpInv**, **RpBloqueInv**).
- Nodo **O** a **S**: respuesta con bloque (**RpBloque**), respuesta con o sin bloque confirmando fin de invalidación (**RpInv**, **RpBloqueInv**).

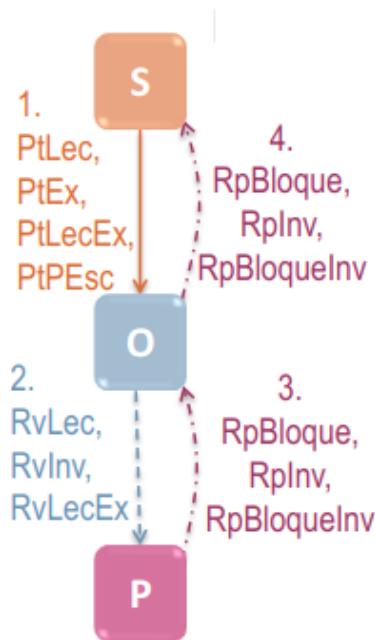


Figura 52: MSI con directorios sin difusión.

**MSI con directorios con difusión** Un bloque en caché puede tener tres estados distintos: **modificado (M)**, **compartido (C)** o **inválido (I)**.

Un bloque en MP puede ser **válido** o **inválido**.

Hay varios tipos de nodos: **solicitante (S)**, origen (O), modificado (M), propietario (P) y compartidor (C).

Difusión de petición del nodo S a:

- **O** y **P**: lectura de un bloque (**PtLec**), lectura con acceso exclusivo (**PtLecEx**), petición de acceso exclusivo sin lectura(**PtEx**).
- **O**: posescritura (**PtPEsc**).

Respuesta de:

- Nodo **P** a **O**: respuesta con bloque (**RpBloque**) o respuesta con/sin bloque confirmando invalidación (**RpInv**, **RpBloqueInv**).
- Nodo **O** a **S**: respuesta con bloque (**RpBloque**), respuesta con o sin bloque confirmando fin de invalidación (**RpInv**

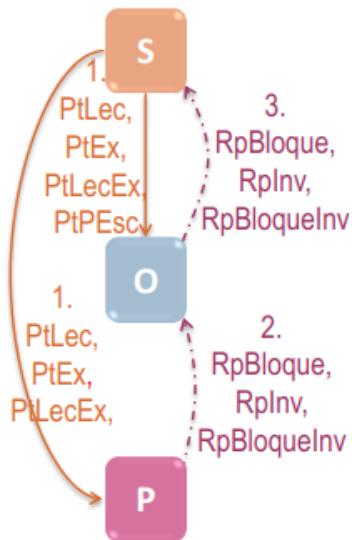


Figura 53: MSI con directorios con difusión.

### 3.3. Consistencia del sistema de memoria

#### 3.3.1. Concepto de consistencia de memoria

Un **modelo de consistencia de memoria** especifica el orden en el cual las operaciones de acceso a memoria (lectura/escritura) deben *parecer* haberse realizado.

La coherencia sólo abarca operaciones realizadas por múltiples componentes (procesos o procesadores) sobre una misma dirección.

#### 3.3.2. Consistencia secuencial

En un sistema uniprocesador, este orden es el orden secuencial ejecutado por el programador, denominado **orden del programa**. Sin embargo, el hardware o el compilador pueden alterar el orden de las instrucciones para mejorar prestaciones, pero debe parecer que no se ha alterado para garantizar el modelo de consistencia al programador. En concreto, se debe garantizar que:

- Cada lectura de una dirección proporcione el último valor escrito en esa dirección (dependencia RAW).
- Si se escribe varias veces en una dirección se debe retornar el último valor escrito (dependencia WAW).
- Si se escribe en una dirección que ha sido previamente leída, no se debe obtener en la lectura previa el valor escrito posteriormente (dependencia WAR).
- No se puede escribir en una dirección si la escritura depende de una condición que no se cumple (dependencias de control).

El **modelo de consistencia secuencial (SC)** requiere que todas las operaciones de memoria parezcan ser ejecutadas una cada vez (ejecución *atómica*) y que todas las operaciones de un único procesador o proceso parezcan ejecutarse en el *orden descrito* por el programa de entrada al procesador.

El modelo de consistencia secuencial es el que suele esperar el programador a alto nivel, ya que presenta el sistema de memoria como una memoria global conectada a todos los procesadores a través de un conmutador central.

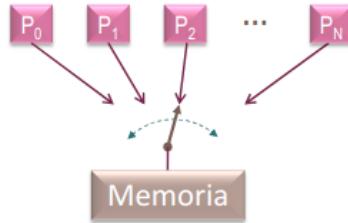
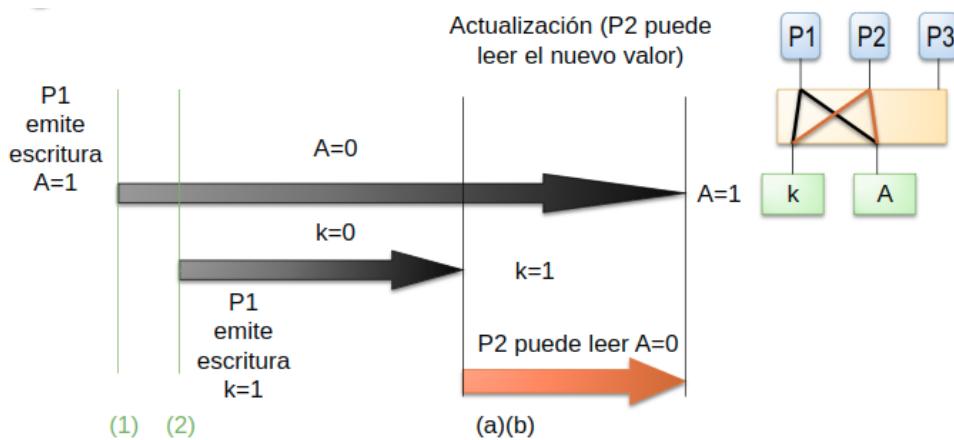


Figura 54: Presentación del sistema a los programadores mediante el modelo SC.

**Ejemplo** Supongamos que disponemos del siguiente código:

```

1 P1{
2     A=1;           //Escribir A
3     K=1;           //Escribir K
4 }
5 ****
6 P2{
7     while(k==0){}      //Leer K
8     var=A;           //Leer A
9 }
```



No se garantiza el orden  $W \rightarrow W$

Figura 55: Ejemplo de ejecución con SC.

### 3.3.3. Modelos de consistencia relajados

Los modelos de memoria relajados difieren en cuanto a los requisitos para garantizar consistencia secuencial (SC) relajan:

- **Orden del programa.** Pueden permitir que en el código ejecutado en un procesador se relaje el orden entre dos accesos a distintas direcciones. Pueden permitir relajar distintos órdenes:  $W \rightarrow W, W \rightarrow R, R \rightarrow W$  o  $R \rightarrow R$ .
- **Atomicidad.** Hay modelos que permiten que un procesador pueda leer el valor escrito por otro procesador antes de que la escritura se haga visible al resto.

Los modelos relajados de consistencia hardware comprenden:

- Especificaciones sobre los órdenes de acceso a memoria que no garantiza el sistema de memoria (tanto órdenes de un mismo procesador como atomicidad en las escrituras).
- Mecanismos que ofrece el hardware para forzar de forma explícita un orden (que no esté garantizado) cuando sea necesario.

Modelo	Orden del programa relajado $W \rightarrow R$ $W \rightarrow W$ $R \rightarrow RW$			Orden global Lec. anticipada propia de otro	Instrucciones para garantizar los órdenes relajados por el modelo
Sparc-TSO, x86-TSO	Si			Si	I-m-e (instruc. lectura-modificación-escritura atómica)
Sparc-PSO	Si	Si		Si	I-m-e, STBAR (instrucción STore BARrier)
Sparc-RMO	Si	Si	Si	Si	MEMBAR (instrucción MEMory BARrier)
PowerPC	Si	Si	Si	Si	SYNC, ISYNC (instrucciones SYNChronization)
Itanium	Si	Si	Si	Si	LD.ACQ, ST.REL, MF (ACQuisition LoaD, RElease STore, Memory Fence), y cmpxchg8.acq y otras I-m-e
ARMv7	Si	Si	Si	Si	DMB (Data Memory Barrier)
ARMv8	Si	Si	Si	Si	LDA   LDAR, STL   STLR (LoaD-Acquire, STore-reLease 32b 64b), LDAEX   LDAXR, STLEX   STLXR (LoaD-Acquire eXclusive, Store-reLease eXclusive 32b 64b), DMB

Figura 56: Ejemplos de modelos relajados.

#### Modelos que relajan $W \rightarrow R$

Estos modelos permiten que las lecturas adelanten a las escrituras, pero evitando problemas de dependencias RAW. Por tanto, eliminan la ordenación  $W \rightarrow R$ , lo que permite ocultar las latencias de escritura. Con esta relajación la mayor parte de los códigos SC funcionan correctamente. Cuando sea necesario se podrán usar instrucciones específicas de serialización.

Estos modelos los implementan procesadores que utilizan buffer de escritura FIFO para permitir que los accesos a memoria de escritura no retarden la ejecución del código bloqueando lecturas posteriores. Además, de forma general, permiten que el procesador pueda obtener directamente el valor de una lectura del buffer de escritura, permitiendo que el procesador pueda leer antes que el resto una escritura propia.

Al utilizar estos buffers, las escrituras de un procesador no se verán inmediatamente por otros procesadores. Pero, cuando se vean, lo harán respetando el orden del programa.

Hay sistemas en los que se permite que un procesador pueda leer la escritura de otro antes que el resto (acceso no atómico). Para garantizar acceso atómico se pueden usar instrucciones de lectura/escritura/modificación atómicas.

### Modelos que relajan $W \rightarrow R$ y $W \rightarrow W$

Estos modelos, además, relajan el orden entre escrituras (a distintas direcciones), eliminando por tanto los órdenes  $W \rightarrow R$  y  $W \rightarrow W$ . Así, por ejemplo, estos modelos permiten que el hardware (como la red de interconexión) solape escrituras a memoria a distintas direcciones, de forma que puedan llegar a la memoria o a cachés de otros procesadores fuera del orden del programa.

En sistemas con este modelo se proporciona hardware para garantizar los dos órdenes (por ejemplo, *Sun Sparc*).

Hay casos en los que estos modelos no se comportan como SC. Por ejemplo:

```
1 P1{
2     A=1;          //Escribir A
3     K=1;          //Escribir K
4 }
5 ****
6 P2{
7     while(k==0){}    //Leer K
8     var=A;          //Leer A
9 }
```

### Modelos que relajan cualquier reordenación entre escrituras y lecturas

Estos modelos relajan todos los órdenes ( $W \rightarrow W$ ,  $W \rightarrow R$ ,  $R \rightarrow W$  y  $R \rightarrow R$ ). Se cambia el orden siempre que no haya conflictos entre los accesos implicados (es decir, se mantiene la consistencia en el procesador evitando problemas por dependencias de datos o control).

Dentro de este grupo se encuentra, el **modelo de ordenación débil** y el de **consistencia de liberación**.

El **modelo de ordenación débil** se basa en mantener el orden entre accesos sólo en los puntos de sincronización del código. Este modelo tiene en cuenta que cuando se necesita coordinar el acceso a una variable compartida (para permitir la comunicación entre procesos) se añade código extra de sincronización.

Las operaciones de memoria se clasifican en dos: de datos y de sincronización. Para forzar el orden entre dos operaciones, el programador debe etiquetar alguna de ellas como operación de sincronización. Si  $S$  es una operación de sincronización (liberación o adquisición), el modelo garantiza los siguientes órdenes:

- Las operaciones de acceso a memoria anteriores en el orden del programa a una operación de sincronización deben completarse antes de ésta ( $WR \rightarrow S$ ).

- Una operación de sincronización se debe completar antes que las operaciones de acceso a memoria posteriores ( $S \rightarrow WR$ ).

```

1   for(i=iproc; i<n ; i+=nproc)
2       sump+=a[i];
3       lock(k);           //1 -> Adquisición. (.isync) (membar) (dmbr)
4           sum+=sum;        //2 -> SC
5       unlock(k);         //3 -> Liberación.      (sync) (membar) (dmbr)

```

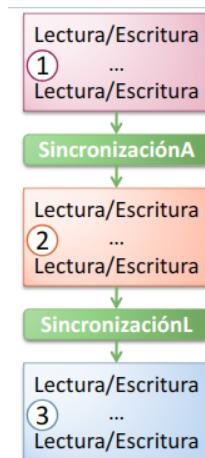


Figura 57: Ejemplo de modelo de ordenación débil.

El **modelo de consistencia de liberación** además tiene en cuenta que hay dos tipos de código utilizado en el proceso de sincronización: el que se ejecuta para adquirir el acceso por un único proceso y el que se ejecuta para que un proceso de permiso a otro para acceder a las variables o recursos compartidos. Por tanto, este modelo distingue entre dos tipos de operaciones de sincronización: **adquisición** y **liberación**.

Si SA es una operación de adquisición y SL de liberación, este modelo mantiene los siguientes órdenes:

- Entre una operación de adquisición y cualquier otra posterior ( $SA \rightarrow WR$ ).
- Entre cualquier operación de acceso a memoria y una operación de liberación posterior ( $WR \rightarrow SL$ )

```

1   for(i=iproc; i<n ; i+=nproc)
2       sump+=a[i];
3       lock(k);           //1 -> Adquisición. (.acq) (lda)
4           sum+=sum;        //2 -> SC
5       unlock(k);         //3 -> Liberación.      (.rel) (stl)

```

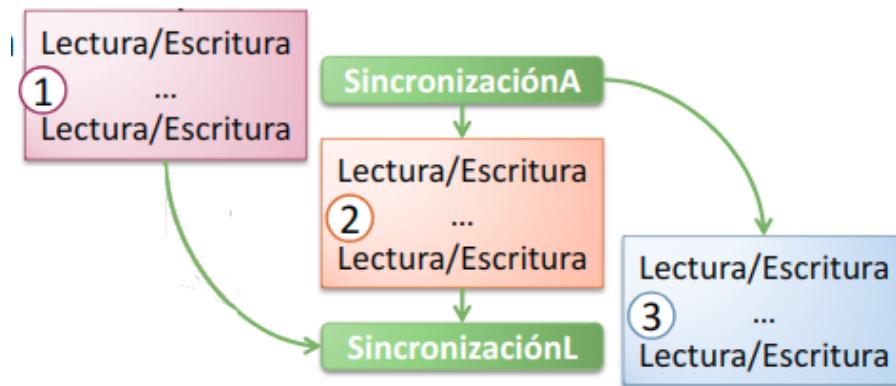


Figura 58: Ejemplo de modelo de consistencia de liberación.

### 3.4. Sincronización

#### 3.4.1. Comunicación en multiprocesadores y necesidad de usar código de sincronización

La comunicación entre procesos en multiprocesadores se realiza como en los sistemas uniprocesador, a través de la memoria compartida. Para poder implementar de forma efectiva la comunicación entre procesos a través de variables compartidas, se necesita sincronizar el acceso a las variables compartidas entre los procesos del grupo que intervienen en la comunicación.

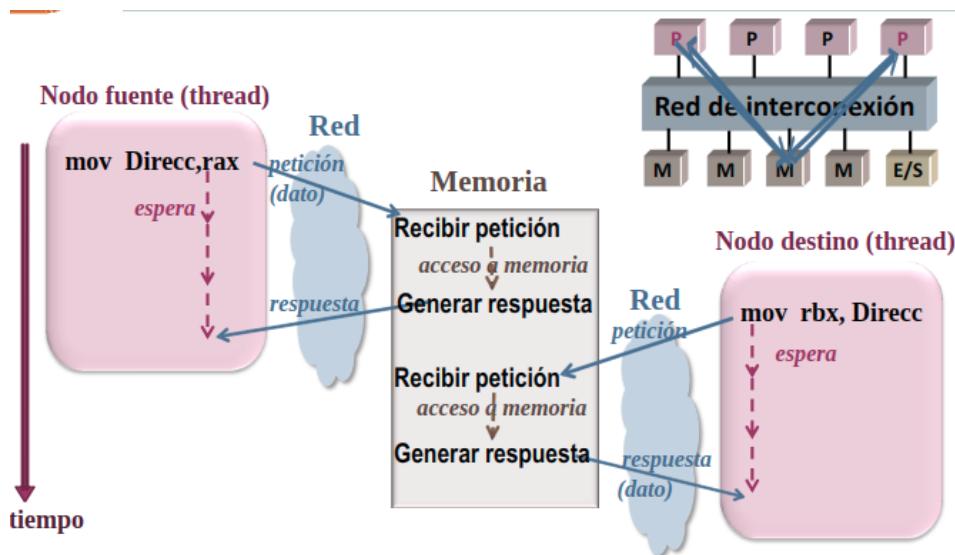


Figura 59: Comunicación en un procesador.

- **Comunicación uno-a-uno.** Se deben sincronizar los dos procesos que intervienen para garantizar que el proceso que recibe (leyendo de una variable compartida) no acceda a ella antes de que el proceso emisor escriba el dato. Además, si la variable se utiliza varias veces (por ejemplo en un bucle), hay que garantizar que no se envía un nuevo dato a través de la variable hasta que no se haya recibido el anterior. En definitiva, se necesita un mecanismo que garantice la exclusión mutua (sólo un proceso puede acceder en un momento dado a una dirección compartida). Una **sección crítica** es una secuencia de instrucciones que se deben ejecutar en exclusión mutua.

- **Comunicación colectiva.** En este caso hay que coordinar el acceso de múltiples procesos a una variable compartida, de forma que escriban uno tras otro (sin interferencias entre ellos) o lean cuando tengan disponibles los resultados definitivos. Puede haber uno o varios procesos emisores y uno o varios receptores. No obstante, para garantizar que no interfieran se necesita un acceso en **exclusión mutua**. Además debe garantizarse que no se accede al resultado hasta que todos los involucrados hayan ejecutado su sección crítica.

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

Figura 60: Comunicación uno-a-uno.

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

Figura 61: Comunicación colectiva.

En el ejemplo de comunicación colectiva, el acceso a *sum* (lectura-modificación-escritura) se debe realizar en exclusión mutua. Para ello podemos usar *cerrojos*. Además, el proceso 0 no debería imprimir hasta que no haya terminado el bucle. Para ello, usaremos *barreras*.

### 3.4.2. Soporte software y hardware para sincronización

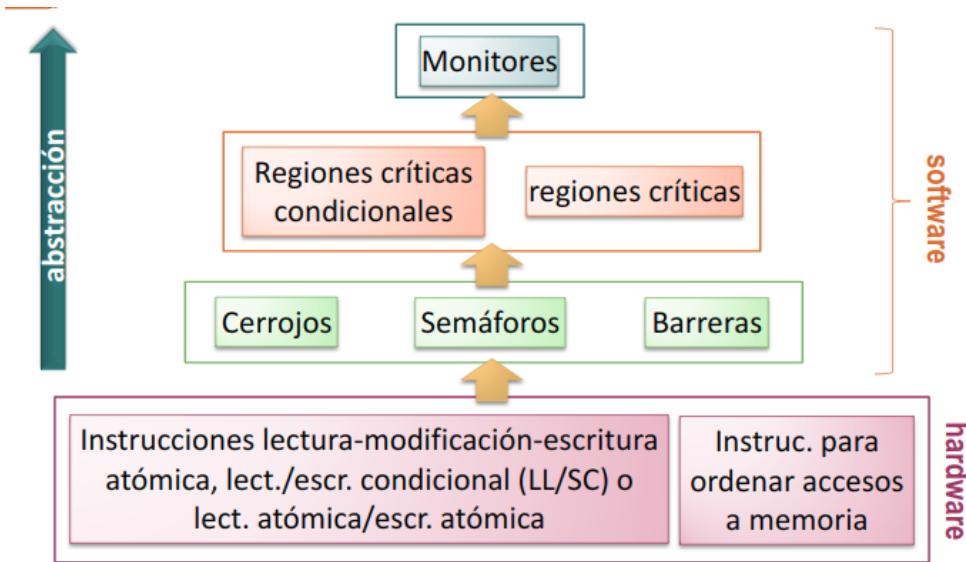


Figura 62: Soporte software y hardware para sincronización.

Se pueden utilizar varias primitivas para realizar la lectura-modificación-escritura atómica:

- **Test&Set.** Realiza de forma atómica sobre una variable compartida  $x$  las siguientes acciones:
  1. Lee en una variable local el contenido actual de  $x$ .
  2. Escribe un 1 en  $x$ .
  3. Devuelve el contenido de la variable local (anterior valor de  $x$ ).

Se puede implementar con una instrucción de intercambio ( $xchg$ ).

- **Intercambio.** Es más general que Test&Set, ya que permite almacenar cualquier valor en la variable compartida. Se puede implementar con  $xchg$ .

- **Fetch&Operation** (Fetch&Add, Fetch&Or...):
  1. Lee en una variable local el contenido de  $x$  (valor que devolverá).
  2. Escribe en  $x$  el resultado de realizar la operación entre  $x$  y el otro valor pasado.

Se puede implementar con la instrucción  $lock xadd$ .

- **Compare&Swap.**

1. Lee el contenido de  $x$ .
2. Si coincide con  $a$ , intercambia  $x$  con  $b$

Se puede implementar mediante la instrucción  $lock cmpxchg$ .

```

1 Test&Set(x){
2     temp=x;
3     x=1;
4     return temp;
5 }
```

Figura 63: Test&Set

```

1 Fetch&Add(x,a){
2     temp=x;
3     x=x+a;
4     return temp;
5 }
```

Figura 64: Fetch&Add

```

1 Compare&Swap)(a,b,x){
2     if(a==x)
3         swap(x,b);
4 }
```

Figura 65: Compare&Swap

### 3.4.3. Cerrojos

Los cerrojos proporcionan una forma de asegurar exclusión mutua, es decir, que un sólo proceso acceda en un momento dado a una variable debido a que se encuentra en sección crítica. Utilizan dos funciones para sincronizar:

- **Cierre del cerrojo, lock(k).** Un proceso intenta adquirir el derecho a acceder a una sección crítica. Si varios la intentan, solo uno debe adquirirla mientras que los otros deben esperar. Todos los procesos que ejecuten *lock()* mientras el cerrojo esté cerrado deben quedar esperando.
- **Apertura del cerrojo, unlock(k).** Libera a uno de los procesos que se encuentran esperando a una sección crítica. En caso de no haber procesos en espera, permite que el siguiente proceso que solicite el cierre lo adquiera y no tenga que esperar.

```

1 for(i=ithread ; i<n; i+=ithread)
2     sump+=a[i];
3 lock(k);           //Adquiero EM
4 sum+=sump;          //SC
5 unlock(k);         //Libero EM
```

Figura 66: Ejemplo de cerrojo

Hay dos posibles implementaciones del mecanismo de espera:

- **Espera ocupada.** El proceso se queda ejecutando un ciclo en el que consulta constantemente si se le permite pasar.
- **Bloqueo.** El proceso queda suspendido, dejando el procesador a otro proceso.

### Cerrojo simple

Se puede implementar con una variable compartida  $k$  que tome dos valores: abierto(0) o cerrado(1). La apertura (**unlock**) escribirá un 0 y el cierre (**lock**) escribirá un 1. Estas operaciones deben ser **atómicas**.

### Cerrojos con etiqueta

Estos cerrojos fijan un orden FIFO para que los procesos adquieran el cerrojo. Por tanto, el primer proceso que solicite el cerrojo será el primero en obtenerlo.

#### 3.4.4. Barreras

Se utilizan para sincronizar entre sí, en algún punto del código, los procesos que colaboran en la ejecución de un trozo del mismo. De esta forma podemos garantizar que ningún proceso sobrepasa un punto determinado del código hasta que todos lleguen ahí.

La barrera irá contando todos los procesos que vayan llegando. Una vez que este valor sea igual al número de procesos, los liberará.

#### 3.4.5. Apoyo hardware a primitivas software

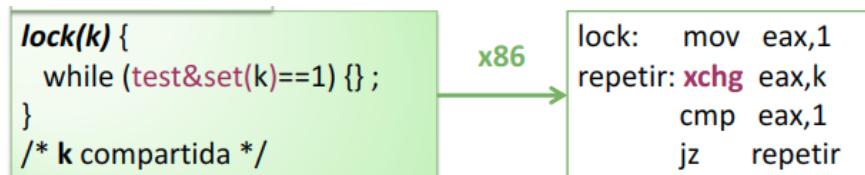


Figura 67: Cerrojos con Test&Set.

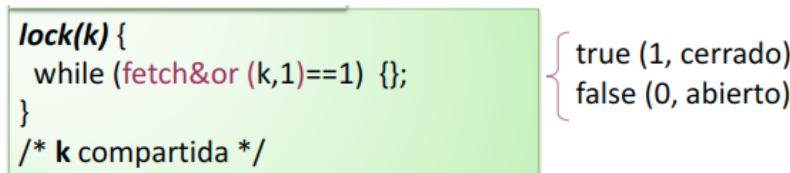


Figura 68: Cerrojos con Fetch&Or.

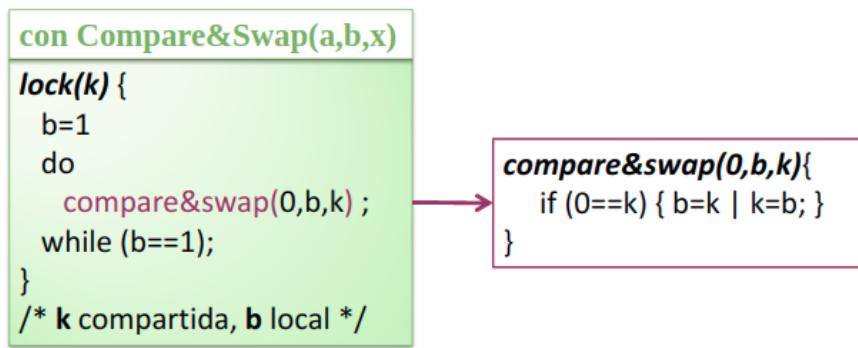


Figura 69: Cerrojos con Compare&Swap.

```

1 for(i=ithread ; i<n; i+=ithread)
2     sump+=a[i];
3 fetch&add(sum,sump);

```

Figura 70: Suma con Fetch&Add.

```

1 for(i=ithread ; i<n; i+=ithread)
2     sump+=a[i];
3 do{
4     a=sum;
5     b=a+sump;
6     compare&swap(a,b,sum);
7 while(a!=b);

```

Figura 71: Suma con Compare&Swap.

### 3.5. Ejercicios

1. En un multiprocesador NUMA con 16nodos 4GBytes por nodo , y líneas de cache de 64 Bytes. ¿Cuántas entradas tiene el directorio de memoria utilizado en cada nodo para mantener la coherencia de cache en un protocolo MSI sin difusión?

#### Solución

$$4GB = 2^2 \cdot 2^{30}B = 2^{32}B$$

$$64B = 2^6B$$

$$\frac{2^{32}B}{2^6B} = 2^{26} \text{ entradas}$$

2. En el multiprocesador NUMA de la pregunta anterior, ¿ cuántos bits tiene cada entrada del directorio que se usa para mantener la coherencia de caché?

#### Solución

$$16 \text{ nodos} + 1 \text{ bit}_{\text{estado}} = 17 \text{ bits}$$

3. ¿Qué valores puede tomar R si el modelo de consistencia del computador no respeta el orden  $W \rightarrow W$  (los demás sí) e inicialmente X=Y=0?

```

1   P1{
2       X=2; // (a)
3       Y=1; // (b)
4   }
      1   P2{
2           R=1; // (c)
3           if(Y==1) R=X; // (d)
4       }

```

### Solución

Los posibles valores son 0,1 y 2. Veamos cuales se pueden obtener:

- 0: a→d→c→b.
- 1: c→a→d→b.
- 2: a→b→c→d.

Por tanto, se cumplen todos.

Si se respetara  $W \rightarrow W$ , los valores posibles serían 1 y 2.

4. ¿Qué valor deben tener r1,r2 y r3 para que este código implemente la función lock(k) de un cerrojo? (0=abierto, 1=cerrado).

```

1 lock(k){
2     b=r1;
3     do{
4         compare&swap(r2,b,k); /si r2==k, k y b se intercambian.
5     while(b==r3);

```

### Solución

Comenzamos la deducción.

- Si vamos a cerrar el cerrojo, debe estar abierto (0). Es decir, en compare&swap el valor del cerrojo debe ser 0. Por tanto, **r2=0**.
- Para cerrarlo, debemos asignar el valor 1. Fijandonos de nuevo en compare&swap vemos que **b=1 → r1=1**.
- Por último, la comprobación del bucle. Se debe esperar (espera ocupada) mientras que el cerrojo no cambie su estado (b se intercambie con k). Por tanto, como **b=1 → r3=1**.