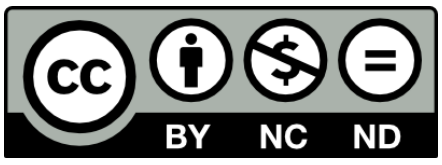
# Arquitectura de Computadores


**Los Del DGIIM**, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



# Arquitectura de Computadores

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

# Índice general

# 1. Arquitecturas TLP

En este capítulo, nos centraremos en arquitecturas que permiten ejecutar de forma paralela o concurrente múltiples flujos de instrucciones (o *threads*) que comparten memoria. Se tratan de arquitecturas con paralelismo a nivel de *thread* (*Thread-Level Parallelism*) con una única instancia del Sistema Operativo. Por tanto, cada vez que mencionemos arquitecturas TLP, nos estamos refiriendo a arquitecturas TLP con una única instancia del Sistema Operativo. En este contexto, el SO es el encargado de gestionar los flujos de instrucciones.

Los paradigmas de programación paralela por variables compartidas son los más fáciles de implementar en este tipo de arquitecturas, mientras que las orientadas a paso de mensajes están más relacionadas con arquitecturas TLP con múltiples instancias del SO.

La compartición de memoria que se da trae conceptos como la coherencia del sistema de memoria, consistencia del sistema de memoria o la sincronización entre flujos; conceptos que estudiaremos a lo largo de este capítulo.

## 1.1. Tipos de Arquitecturas

### 1.1.1. Objetivos

Esta sección está orientada a:

- Distinguir entre cores multhread, multicores y multiprocesadores.
- Comparar entre cores multithread de grano fino, grueso y cores con multithread simultáneo.

### 1.1.2. Clasificaciones de arquitecturas TLP

Las arquitecturas TLP con una instancia del SO pueden clasificarse en:

**Multiprocesadores.** Son capaces de ejecutar en paralelo varios flujos de instrucciones (hilos) en un computador con varios núcleos o procesadores de forma que cada flujo se ejecuta en un núcleo o procesador distinto.

Podemos encontrarnos multiprocesadores en un chip (como los multinúcleos), en una placa o en uno o varios armarios.

**Multinúcleos (*multicores*).** Pueden ejecutar en paralelo varios flujos de instrucciones en un chip de procesamiento con múltiples núcleos de forma que cada flujo se ejecuta en un núcleo distinto. Un chip multinúcleo no es más que un multiprocesador en un chip.

La denominación de *multicores* proviene de un nombre comercial que dio Intel a sus multiprocesadores en un chip. Además, denominó *procesador* a los chips o encapsulados de procesamiento y *núcleos* (o *cores*) a los procesadores.

**Núcleos (o cores) *multithread*.** Se trata de un núcleo de procesamiento (un procesador) en el que se ha modificado su arquitectura ILP (*Instruction Level Parallelism*) para poder ejecutar flujos de instrucciones de forma concurrente o en paralelo.

### 1.1.3. Repaso de arquitecturas ILP

Recordamos lo que eran las arquitecturas ILP (*Instruction Level Parallelism*): son la capacidad por parte de un procesador de ejecutar múltiples instrucciones en paralelo para mejorar el rendimiento del sistema. Las arquitecturas se centran en identificar y aprovechar las dependencias de datos entre las instrucciones para ejecutarlas de forma eficiente.

#### Etapas de ejecucución

Antes de continuar, recordamos las etapas básicas para la ejecución de una instrucción:

**Etapa de captación de instrucciones (*Instruction Fetch*).**
Etapa en la que se capta de la caché de instrucciones la siguiente instrucción a ejecutar, incrementando el valor del PC(*Program Counter*).

**Etapa de decodificación de instrucciones (*Instruction Decode*).**
Etapa en la que se decofica la instrucción captada para determinar su tipo y operaciones a realizar. Se identifican aquí las dependencias de datos y condiciones de control que pueden afectar la ejecución de la instrucicón.

**Etapa de ejecución (*Execution*).**
Se lleva a cabo la ejecución de la instrucción. Podemos encontrarnos 4 tipos de instrucciónes (en relación a ellos se ejecutará una cosa u otra).

- Operaciones de enteros.
- Operaciones de coma flotante.
- Saltos.
- Escrituras o lecturas de memoria.

Aunque esta última no se realizaría en esta etapa, es importante para el desarrollo que vamos a hacer de arquitecturas ILP.

**Etapa de acceso a memoria (*Memmory*).**
Se accede a memoria en caso de que sea necesario (depende de la instrucción).

**Etapa de almacenamiento de resultados en registros (*Write-Back*).**
Se almacenan en los registros del procesador los resultados de la instrucción, si es necesario.

**Formas de arquitecturas ILP**

Podemos encontrarnos con dos formas principales de paralelizar estas etapas a la hora de desarrollar una estructura ILP:

**Escalar segmentada.**
Segmentaremos las etapas de ejecución de forma que dispongamos de 5 módulos que sean capaces de procesar su parte de forma paralela. Incluiremos *buffers* auxiliares entre las distintas etapas.

**VLIW y superescalar.**
Consideraremos simplemente 4 etapas (fusionaremos las de ejecución y memoria en una), de forma que replicaremos los componentes de la unidad funcional para que esta sea capaz de ejecutar al mismo tiempo diversos tipos de instrucciones. También es necesario el uso de *buffers* auxiliares.

Por ejemplo, podemos replicar los componentes de la unidad funcional de forma que podamos ejecutar en paralelo (al mismo tiempo):

- Operaciones con enteros.
- Operaciones de coma flotante.
- Operaciones de lecturas y escrituras en memoria.
- Saltos.

Podemos además tener no sólo una unidad sino varias de las ya mencionadas (2 unidades para operaciones con enteros, ...).

Además, las 3 etapas restantes estarán segmentadas. Podemos tener también unidades superescalares en las que tengamos replicadas además las etapas de captación, decodificación y *write-back*.

### 1.1.4. Clasificaciones de cores multithread

Ahora vamos a clasificar los *cores multithread*, que no dejan de ser procesadores con arquitectura ILP que se aprovechan para ejecutar a la vez distintos hilos del sistema operativo de forma concurrente o paralela.

***Temporal Multithreading* (TMT)**
Ejecutan distintos hilos de forma concurrente en el mismo core. De esta forma, emite instrucciones de un único hilo en cada ciclo. Podemos pensar que el core se está multiplexando.

La conmutación entre hilos la decide el hardware.

***Simultaneous MultiThreading* (SMT)**
Ejecuta distintos hilos de forma paralela en el mismo core. Pueden llegar a

emitir en un sólo ciclo instrucciones de varios hilos. Para llevar esto a cabo, necesitamos un core superescalar.

No implementa conmutación entre hilos, al no ser necesaria.

Según qué tan seguido intercambiamos los hilos del core, nos encontramos con:

**TMT de grano fino (*Fine-grain multithreading*, FGMT).**
La conmutación de hilos en el core se realiza en cada ciclo. Presentan un coste de cambio de contexto bajo, no es necesario perder ningún ciclo para realizar los cambios de contexto.

La planificación del siguiente hilo a ejecutar puede ser *round-robin* u otra técnica de planificación (podemos guiarnos por el hilo menos recientemente ejecutado, por accesos a datos, por saltos no predecibles, por operaciones con gran latencia, ...).

**TMT de grano grueso (*Coarse-grain multithreading*, CGMT).**
La conmutación entre hilos no se realiza en cada ciclo. Presentan un mayor coste por cambios de contexto: pueden perderse entre ninguno y varios ciclos debido a los cambios de contexto.

La planificación puede depender cualquier técnica, como tras intervalos de tiempos prefijados (*timeslice multithreading*), por eventos de cierta latencia (*switch-on-event multithreading*), ...

### Clasificación de cores con CGMT con conmutación por eventos

Podemos conmutar los hilos del core de grano grueso de forma:

**Estática.**
Realizando la conmutación de forma *explícita*, mediante nuevas instrucciones para conmutación añadidas al repertorio; o de forma *implícita*, al detectar instrucciones e carga, almacenamiento, salto, ...

- Como ventaja, destacamos el bajo coste de los cambios de contexto (de 0 o 1 ciclos).
- Como inconveniente, pueden producirse cambios de contexto innecesarios.

**Dinámica**
La conmutación se realiza típicamente por fallos de caché o por interrupciones (interrupciones por señales).

- Como ventaja, reduce los cambios de contexto innecesarios de la estática.
- Como inconveniente, la sobrecarga que se añade por los cambios de contexto es mayor.

| Hardware | CGMT | FGMT |
|---|---|---|
| Registros | Replicado (al menos el PC) | Replicado |
| Almacenamiento | Multiplexado | Cualquiera de las 4 |
| Hardware de etapas del cauce | Multiplexado | Captación repartida o compartida, el resto multiplexadas |
| Necesidad de distinguir el hilo de una instruc. | Sí | Sí |
| Hardware para conmutar | Sí | Sí |
| Hardware | SMT | CMP |
| Registros | Replicado | Replicado |
| Almacenamiento | Todo menos multiplexado | Replicado |
| Hardware de etapas del cauce | Unidad funcional compartida, el resto repartidas o compartidas | Replicado |
| Necesidad de distinguir el hilo de una instruc. | Sí | No |
| Hardware para conmutar | No | No |

Tabla 1.1: Comparación de tipos de multithreading.

**Característica**

En un núcleo multithread, podemos usar las siguiente modificaciones con el objetivo de ejecutar varios hilos en un mismo core:

- Multiplexado: Hacemos que los hilos se turnen en el uso de una unidad.
- Repartición: Repartimos una unidad entre (al menos) dos hilos, de forma que a uno le asociamos su zona y al otro la suya.
- Compartición: Hay (al menos) dos hilos que acceden a la misma unidad de forma simultánea.
- Replicación: Disponemos de varias unidades, de forma que cada hilo puede hacer uso de una.

Notemos que tanto el precio de implementación como la bondad de la técnica se encuentran en orden creciente (siendo más caro y mejor realizar la replicación de unidades).

Una vez desarrollada la clasificación de cores multithread, podemos mostrar la Tabla 1.1 a modo de resumen, que nos ayudará a entender mejor cada tipo de multithreading. "CMP" es un Chip multicore, aquel en el que tenemos replicado todo el cauce (esto es, todas las unidades de la etapa de ejecución).

### 1.1.5. Comparativa de cores multithread

ESTOS APUNTES ESTÁN AÚN EN DESARROLLO.

## 1.2. Coherencia del sistema de memoria

A la hora de usar multiprocesadores, surge un problema fundamental de manera natural: las incoherencias en memoria. A lo largo de esta sección definiremos este problema, dando ejemplo y protocolos que lo resuelven. Cabe mencionar que todos los multiprocesadores salvo los NUMA implementan la coherencia del sistema de memoria por hardware.

### 1.2.1. Objetivos

Tras esta sección, debería ser capaz de:

- Comparar los métodos de actualización de memoria principal implementados en caché.
- Comparar las alternativas para propagar una escritura en protocolos de coherencia de caché.
- Explicar qué debe garantizar el sistema de memoria para evitar problemas por incoherencias.
- Descibir las partes en las que se puede dividir el análisis o el diseño de protocolos de coherencia.
- Distinguir entre protocolos basados en directorios y protocolos de espionaje (snoopy).
- Explicar el protocolo de mantenimiento de coherencia de espionaje MSI.
- Explicar el protocolo de mantenimiento de coherencia de espionaje MESI.
- Explicar el protocolo de mantenimiento de coherencia MSI basado en directorios con difusión y sin difusión.

### 1.2.2. Definición del problema

La utilización de memoria caché trae consigo el esquema de jerarquía de memoria y la posibilidad de tener en distintas jerarquías una misma posición de memoria repetida. En sistemas uniprocesador, si tratamos de modificar una posición de memoria, esta se llevará a caché y será modificada en caché, indicando que ha sido modificada. En un cierto momento, la modificación en esta jerarquía de caché será comunicada a jerarquías de memoria mayores, hasta llegar a memoria principal y modificar dicho dato. De esta forma, la próxima vez que se necesite en caché dicha posición, estará actualizada conforme a la última modificación.

Recordamos ahora que cada procesador lleva asociada una memoria caché y, al ser mayor la memoria principal que la caché, necesitamos almacenar en algún sitio qué bloque de memoria principal contiene cada entrada de memoria caché. Esta información se almacena en una tabla asociada a la caché. En dicha tabla hay una entrada por cada dirección de la memoria caché y, en cada entrada, se almacena

| | | |
|---|---|---|
| 0 | Dirección de MP | Bits |
| 1 | Dirección de MP | Bits |
| | ⋮ | ⋮ |
| $N-1$ | Dirección de MP | Bits |

Tabla 1.2: Tabla para una caché de $N$ direcciones de memoria.

qué bloque de memoria principal está cargado y unos bits de información sobre el bloque cargado. Representamos una aproximación a esta tabla en la Figura 1.2. En un sistema uniprocesador, nos basta con un bit sucio (que indique si el bloque ha sido modificado o no, para saber si hay que escribir en jerarquías supsriores) y un bit de validez (que indique si el bloques es válido).

En un multiprocesador, cada procesador lleva consigo una memoria caché, de forma que todos los procesadores comparten el espacio de memoria. Puede suceder que dos procesadores distintos trabajen con la misma posición de memoria $k$ (y por tanto, tengan al bloque que contiene a $k$ en sus respectivas cachés). Si uno de los dos procesadores decide modificar $k$, se modificará la dirección de memoria caché correspondiente, pero el bloque en la caché del otro procesador y en memoria principal quedarán inalterados. Nos acabamos de encontrar con una incoherencia en el sistema de memoria.

Concretando las ideas, una **incoherencia en el sistema de memoria** se produce cuando en el sistema de memoria las copias de una misma dirección no tienen el mismo contenido. Como hemos ya comentado, en sistemas uniprocesdores teníamos este problema: podíamos tener un bloque en memoria caché modificado que no estuviese modificado en memoria principal. En este caso hablamos de incoherencias en distintas jerarquías de memoria. Sin embargo, ahora en multiprocesadores podemos tener incoherencias en la misma jerarquía de memoria, tal y como mencionábamos en el ejemplo anterior.

### Métodos de actualización de memoria principal

Las situaciones de incoherencia se deben abordar no permitiendo que se produzcan nunca o bien evitando que causen problemas (que algún componente lea el valor no actualizado de la memoria) en caso de permitirse. Las cachés implementan dos métodos de actualización de memoria principal:

**Escritura directa (*write-through*).**
Con escritura directa, no se permiten situaciones de incoherencia entre caché y memoria principal al escribir en caché: cada vez que se escribe en un bloque de la caché, el correspondiente bloque de la memoria principal es modificado.

Por tanto, tenemos una escritura en memoria principal por cada escritura, lo que requiere usar la red de comunicación entre procesador y memoria principal. Este sobreuso de la red empeora más aún en multiprocesadores, al tener múltiples procesadores que son susceptibles de modificar datos de forma paralela.

Otro problema que se plantea es desaprovechar los principios de localidad espacial y temporal (al modificar una variable, es posible que se modifique otra próxima a ella, o que la ya modificada se vuelva a modificar próximamente), por lo que sería mejor esperar a que termine la modificación en curso (por ejemplo, si estamos iterando sobre un vector y modificando sus componentes), antes de escribir en memoria los cambios modificados.

Cabe destacar también que sí pueden producirse con escritura directa situaciones de incoherencia entre cachés (dos cachés tienen el mismo bloque y una lo modifica) y entre caché y memoria principal (cuando un componente modifica la dirección de un bloque de memoria principal que se encuentra en alguna caché del multiprocesador). En sistemas uniprocesadores, tenemos la ventaja de no necesitar un bit sucio.

**Posescritura (*write-back*).**
En posescritura, cuando una dato es modificado por un procesador, sólo se modifica en la caché correspondiente a dicho procesador. La actualización en memoria principal se produce cuando un bloque modificado (un bloque en el que se ha alterado una dirección de memoria mientras estaba en la caché) es retirado de la caché (por ejemplo, para hacer sitio a otro bloque más necesario). De esta forma, minimizamos el número de accesos a memoria y, por tanto, de uso de la red de conexión entre el procesador y la memoria; aprovechando los principios de localidad espacial y temporal.

De esta forma, se permiten situaciones de incoherencia entre caché y memoria principal (incluso en sistemas uniprocesador). También puede producirse cualquier tipo de incoherencia en este sistema, empeorando la situación que teníamos con escritura directa.

Es necesario además mantener la información sobre qué bloques han sido modificados en caché y cuales no. Esta labor la realiza un bit en la tabla de la caché, llamdo "bit sucio".

Lo usual en cachés es que usen posescritura, debido a sus ventajas frente a escritura directa. A continuación, estaremos hablando siempre de sistemas multiprocesadores, ya que en sistemas uniprocesadores las situaciones de incoherencia ya están resueltas (tanto con escritura directa no permitiendo incoherencias tanto con posescritura, permitiendo incoherencias pero controlando que no provoquen fallos).

### 1.2.3. Protocolos de coherencia entre cachés

Como ya habrás podido deducir, nuestra tarea ahora es buscar cómo resolver las incoherenias ya mencionadas, y manejar las incoherencias que se permiten para que no provoquen fallos en el sistema. Para evitar situaciones de incoherencias entre cachés, se deben cumplir las siguientes dos condiciones:

1. **Propaganción de escrituras.** Se debe garantizar que todo lo que se escribe en la copia de un bloque en caché se propague a las copias del bloque en otras cachés.

2. **Serialización de escrituras.** Se debe garantizar que las escrituras en una dirección se ven en el mismo orden por todos los procesadores: el sistema de memoria debe parecer que realiza en serie las operaciones de escritura en la misma dirección. Debe dar la impresión de que estas operaciones sean atómicas.

   Esto es de vital importancia en arquitecturas donde no todos los procesadores tienen el mismo (o similar) tiempo acceso a sus cachés, como los NUMA, debemos garantizar que el primer procesador que se dispuso a escribir en memoria sea el primero que lo haga.

Los **protocolos de coherencias de caché** buscan resolver el problema de las incoherencias entre cachés, de forma que cada escritura en caché sea visible para el resto de procesadores, propagando de forma fiable el valor escrito en una dirección. Los protocolos de coherencia usan dos alternativas principales para propagar escrituras a otras cachés:

**Escritura con actualización (*write-update*).**
Siempre que se modifique una dirección en la copia de un bloque en una caché, se modifica dicha dirección en las copias del mismo bloque de memoria que tengan el resto de procesadores en sus cachés (en caso de tenerlo). Si se cumplen los principios de localidad espacial y temporal, esto provoca una alta sobrecarga en la red de comunicación, al tener que avisar al resto de procesadores la modificación de una variable.

**Escritura con invalidación (*write-invalidate*).**
Antes de que un procesador modifique un bloque en su memoria caché, invalida el resto de copias del mismo bloque en las cachés de otros procesadores. Posteriormente, es libre de modificar su bloque tantas veces como desee, obteniendo un *acceso exclusivo* al bloque.

Cuando otro procesador quiera acceder a dicho bloque desde su caché, en caso de tenerlo, verá que estará invalidado y deberá solicitarlo a la memoria (en caso de que el bloque se encuentre actualizado en memoria) o al procesador que tenía el acceso exclusivo al bloque. Es necesario por tanto disponer de un bit en la tabla de la caché que indique si un bloque se encuentra invalidado o no.

Notemos que invalidar es más rápido que actualizar, ya que sólo necesitamos recibir la información del bloque invalidado y cambiar un bit, en lugar de reescribir el bloque completo.

Esta práctica sólo permite compartir bloques de memoria mientras sólo se lee del bloque.

Si escribimos varias veces sucesivas sin que otro procesador lea de su bloque correspondiente, con invalidación podemos reducir el número de accesos (transferencias) a la red. Por otra parte, si deseamos que un procesador escriba un dato para que el resto lo lean podría ser más eficiente usar escritura con actualización, que disminuye en este caso los accesos a la red (notemos que con invalidación necesitamos un acceso a la red por cada caché que falle, pero con actualización con un único acceso a la red podemos actualizar todas las cachés). Cabe destacar que esta ventaja de la actualización no se da en arquitecturas que no implementan difusión. Podemos decir

que en general, la política de actualización genera un tráfico innecesario cuando los datos compartidos se leen por pocos procesadores.

La propagación de las actualizaciones o invalidaciones entre los procesadores se puede realizar:

- Con una **difusión** de los paquetes de actualización o invalidación a todas las cachés.
- Con el envío de los paquetes de actualización o invalidación sólo a aquellas cachés con copias del bloque, que son sólo las que necesitan recibirlo. Es decir, realizar un **envío selectivo**.

  Para esta última alternativa, necesitamos mantener una tabla o directorio de memoria, que informe de las cachés que tienen copia de un determinado bloque para poder realizar la comunicación. Esta tabla tendría una entrada por cada dirección de memoria de cada caché y contendría el bloque que contiene, junto con bits de estado.

### 1.2.4. Protocolos de mantenimiento de coherencia

En una arquitectura UMA, podemos utilizar una red bus para las transferencias entre los procesadores y la memoria que comparten. Los buses implementan la difusión de forma natural. De esta forma, todos los paquetes enviados a la red son visibles por todos los componentes conectados al bus, de forma que toos ven las peticiones en el orden en el que se solicitan (dándose las dos condiciones para evitar situaciones de incoherencias). Todo esto hace que en el caso de los multiprocesadores UMA con red bus no sea necesarios mantener información de la cachés con copias de los bloques, pudiendo suprimir incluso el directorio de memoria del que antes se hablaba.

En este tipo de arquitecturas, es común el uso de **protocolos de espionaje** (*snoopy*), ya que todos los componentes pueden ver (espiar) dicho bus. Cada controlador de caché espía los paquetes del bus y actúa en consecuencia (si por ejemplo otro controlador solicita un paquete invalidado del que nuestra caché tiene acceso exclusivo y no está actualizado en memoria, nuestro controlador invalida la respuesta de la memoria y es él quien responde a la petición del paquete, devolviendo el paquete actualizado). Es por ello sencillo implementar protocolos de coherencia en una arquitectura que use buses. Sin embargo, los protocolos de espionaje no escalan bien debido a retardos que introducen otro tipo de redes[1].

En redes en las que la difusión es costosa de implementar o redes que requieren de una gran escalabilidad, se usan **esquemas basados en directorios**, en los que para reducir el tráfico, se envían los paquetes únicamente a las cachés implicadas (cachés con copia del bloque al que se accede). Es necesario por tanto el uso del directorio de memoria. Sin embargo, también obtenemos un cuello de botella al tener que acceder por cada procesador a dicho directorio. Se obtienen mejores prestaciones distribuyendo el directorio entre los módulos de memoria principal, de forma que el

[1] Al tener que esperar a que todos los procesadores reciban el paquete enviado.

subdirectorio de cada módulo mantenga la información sobre sólo los bloques que contiene cada módulo. De esta forma, los diferentes subdirectorios pueden procesar peticiones en paralelo.

Cabe destacar que no son los únicos tipos de protocolos de coherencia, sino que hay también esquemas organizados en **jerarquía**, compuestos de protocolos de espionaje y directorios, que dependen de la red utilizada en cada nivel.

Resumiento toda esta introducción a los protocolos de coherencia realizada, debemos tener claro que para diseñar un protocolo de mantenimiento de coherencia, debemos planificar:

- La política de actualización en memoria principal: escritura directa o posescritura.
- La política de propagación de escrituras entre cachés: escrituras con actualización o con invalidación.
- El comportamiento:
  - Los posibles estados de un bloque en caché y las acciones que el controlador de caché debe realizar ante eventos recibidos.
  - Los posibles estados de un bloque en memoria principal, junto con las acciones que el controlador de memoria principal debe hacer ante eventos.
  - Los paquetes que genera el controlador de memoria principal.
  - Saber relacionar las acciones con eventos y estados.

A continuación, vamos a estudiar cuatro protocolos de mantenimiento de coherencia, dos para multiprocesadores UMA con red bus (MSI y MESI) y dos para multiprocesadores NUMA en una placa (MSI con y sin difusión). Todos ellos usan posescritura y escrituras con invalidación, como cabría esperar.

Los protocolos de espionaje que estudiaremos se basan en la difusión de los paquetes asociados al mantenimiento de coherencia a todas las cachés. Se implementan de forma eficiente en sistemas basados en buses.

### 1.2.5. Protocolo MSI (*Modified-Shared-Invalid*) de espionaje

Describimos a continuación el protocolo MSI, el protocolo con menor número de estados que utiliza posescritura e invalidación. Debe su nombre a cada estado de un bloque en caché.

#### Estados de un bloque en caché

**Modificado (M).**
Un bloque modificado en caché es la única copia del bloque válida en todo el sistema de memoria.

- En caso de que otro módulo solicite este bloque por el bus, el controlador de su caché debe invalidar la respuesta de la memoria principal (la memoria principal siempre intentará responder)[2] y el controlador de caché debe responder con su propio bloque, que es el único válido.
- El controlador de su caché debe escribir dicho bloque en memoria en caso de ser extraído de su caché.
- El controlador de su caché debe invalidar el bloque si le llega por el bus una petición de invalidación del bloque correspondiente.

**Compartido (S).**
Un bloque compartido en caché indica que todas las copias de dicho bloque que pueda haber en la jerarquía de memoria están actualizadas.

- El controlador de su caché debe invalidar el bloque si le llega por el bus una petición de invalidación del bloque correspondiente.

**Inválido (I).**
Un bloque inválido en caché es un bloque que o no está físicamente en caché (esa posición de caché está vacía) o que está invalidado (por progragación de invalidaciones desde otra caché).

- Si el procesador de la caché accede a un bloque en estado inválido, enviará un paquete de petición del bloque por el bus de la red.

Podemos ordenar estos estados en relación al grado de disponibilidad del bloque por parte del procesador al que pertenece la caché en orden creciente por:

- Inválido.
- Compartido.
- Modificado.

El estado modificado supone que el procesador tiene el uso exclusivo del bloque, de forma que puede disponer de él tanto par aleer como para escribir, sin informar al resto del sistema. Como el módulo que contiene al bloque es el único que dispone de una copia válida del mismo, se trata del encargado de responder a todas las peticiones de este por el bus.

Todos estos estados se implementan añadiendo a la tabla de caché de cada módulo (Tabla 1.2) una serie de bits. Necesitamos representar 3 estados, luego nos será suficiente con disponer de mínimo 2 bits por cada entrada de la tabla. Las implementaciones suelen añadir normalmente los bits de invalidez (indica si el bloque en dicha posición de caché es válido) y sucio (indica si el bloque en dicha posición de caché ha sido modificado) en relación a nuestro modelo de estados MSI. De esta forma:

[2]En caso de estar así implementado; puede haber implementaciones que no realicen esta función, como se verá en breves.

| Estados | Bit de invalidez | Bit sucio | Ambos bits |
|---|---|---|---|
| M | 0 | 1 | 01 |
| S | 0 | 0 | 00 |
| I | 1 | 0 | 10 |

### Estados de un bloque en memoria

Según lo anteriormente explicado, podemos evitar establecer estados para bloques en memoria (habilitando que los controladores de caché puedan inhibir respuestas del controlador de memoria principal). Sin embargo, si se desea almacenar puede hacerse simplemente con dos estados:

**Válido.** El bloque está actualizado en memoria principal y puede haber una copia en una o varias cachés (sin acceso exclusivo).

- El controlador de memoria principal responde con el bloque si ve en el bus una petición con lectura de un bloque que en memoria principal es válido.

**Inválido.** El bloque no está actualiza en la memoria principal y hay una copia válida en alguna caché.

- En caso de que este bloque sea pedido por el bus, será el controlador de caché de la caché correspondiente quien proporcione el bloque.

En caso de desear implementar la validez o no de los bloques en memoria, nos bastaría con disponer de un bit de invalidez por bloque de memoria.

### Paquetes que genera un controlador de caché

Hemos comentado varias veces que los distintos controladores de caché de cada nodo tienen que tener un trabajo en la red de comunicación, pero no hemos especificado cómo se realizan estas comunicaciones.

Un controlador de caché puede generar los siguiente tipos de transferencias como consecuencia de acciones en caché del procesador, o como consecuencia de paquetes recibidos por otros nodos. Puede generar paquetes de petición (comienzan por `Pt`) o de respuesta (comienzan por `Rp`).

**Petición de lectura de un bloque `PtLec(B)`.**
Se genera como consecuencia de una lectura con fallo de caché del procesador del nodo (evento `PrLec`) en relación al bloque `B`. El paquete contendrá la dirección del bloque `B`.

Como respuesta a esta petición, recibirá un paquete de respuesta con el bloque (`RpBloque(B)`) de memoria principal o de la caché que lo tiene en estado modificado, en caso de haberlo.

**Petición de acceso exclusivo a un bloque con lectura de bloque `PtLecEx(B)`.**
Se genera como consecuencia de una escritura con fallo de caché del procesador

(evento `PrEsc`) en un bloque inválido `B` en la caché. El paquete contendrá la dirección del bloque `B`.

Como respuesta se invalidarán las copias del bloque en otras cachés y memoria principal; y se recibirá un paquete de respuesta con el bloque (`RpBloque(B)`) de memoria o de la caché que lo tiene en estado modificado, en caso de haberlo.

**Petición de acceso exclusivo a un bloque `PtEx(B)`.**
Se genera como consecuencia de una escritura del procesador (evento `PrEsc`) en un bloque `B` en estado compartido en la caché (si estuviera en estado modificado, no hace falta generar este paquete). El paquete contendrá la dirección del bloque `B`.

Como respuesta se invalidarán las copias del bloque en otras cachés y memoria principal.

En ciertas implementaciones no existe este paquete; en su lugar se usa el paquete `PtLecEx(B)` descartando el paquete de respuesta (`RpBloque(B)`) que es enviado a través del bus.

**Petición de posescritura de un bloque `PtPEsc(B)`.**
Se genera por el reeemplazo en caché de un bloque `B` en estado modificado, como consecuencia del acceso del procesador a un bloque que no se encuentra en caché. El paquete contendrá la dirección del bloque `B`, así como su contenido.

El procesador no espera ninguna respuesta.

**Respuesta con bloque `RpBloque(B)`.**
Se genera por el controlador de la caché que tiene el bloque `B` solicitado en estado modificado, cuando detecta por el bus una petición `PtLec(B)` o `PtLecEx(B)`.

### Paquetes que genera el controlador de memoria principal

El controlador de memoria principal sólo genera paquetes de respuesta:

**Respuesta con bloque `RpBloque(B)`.**
Se genera como respuesta a paquetes de peticiones de un bloque `B` por `PtLec(B)` o `PtLecEx(B)`.

- Si la memoria guarda el estado de sus bloques de memoria (esto es, los controladores de caché no inhiben la respuesta de la memoria principal), espiará el bus y sólo generará el paquete de respuesta si el bloque `B` se encuentra en estado válido en memoria.
- Si la memoria no guarda el estado de sus bloques de memoria, siempre reponderá con estos paquetes, siendo su respuesta inhibida por el controlador correspondiente.

| Estado | Evento | Acciones del controlador | Estado final |
|---|---|---|---|
| | `PrLec(B)` | | M |
| | `PrEsc(B)` | | M |
| M | `PtLec(B)` | Genera `RpBloque(B)` | S |
| | `PtLecEx(B)` | Invalida copia local y genera `RpBloque(B)` | I |
| | `Reemplazo(B)` | Genera `PtPEsc(B)` | I |
| | `PrLec(B)` | | S |
| S | `PrEsc(B)` | Genera `PtEx(B)` | M |
| | `PtLec(B)` | | S |
| | `PtLecEx(B)` | Invalida copia local de `B` | I |
| | `PrLec(B)` | Genera `PtLec(B)` y recibe `RpBloque(B)` | C |
| I | `PrEsc(B)` | Genera `PtLecEx(B)` invalida en otras cachés y recibe `RpBloque(B)` | M |
| | `PtLec(B)` | | I |
| | `PtLecEx(B)` | | I |

Tabla 1.3: Tabla de acciones del controlador de caché.

### Transiciones de estado de un bloque

En la Tabla 1.3 podemos ver las acciones del controlador de caché de un nodo (los paquetes que genera y los cambios de estado que realiza sobre sus bloques) que provocan los eventos relacionados con un bloque `B`, teniendo en cuenta el estado del bloque en la caché (eventos del procesador o del controlador de caché del nodo, recibidos del exterior a traés de la red en forma de paquete).

**Fallo de lectura.**

El procesaor lee (`PrLec(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de lectura del bloque de memoria `PtLec(B)`.

El estado del bloque en la caché será compartido. La copia del boque en otras cachés también será compartido y en memoria, válido.

`PtLec(B)` provoca los siguientes efectos:

1. Si el bloque se encuentra en otra caché en estado modificado, su controlador deposita en el bus el paquete `RpBloque(B)` y pasa a estado compartido. La memoria también recoge el bloque del bus, pasando a estado válido (si tiene estados).
2. Si el bloque está compartido en otras cachés, la memoria proporciona el bloque a la caché que la solicita. El bloque sigue en estado compartido en cada una de las cachés.

**Fallo de escritura al no estar el bloque en la caché.**

El procesador escribe (`PrEsc(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de acceso exclusivo al bloque de memoria `PtLecEx(B)`.

El estado del bloque en la caché después de la escritura será modificado.

`PtLecEx(B)` provoca los siguientes efectos:

1. Si la memoria tiene el bloque válido, lo deposita en el bus y pasa a estado inválido.
2. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus (invalidando la respuesta de la memoria) y pasa a estado inválido.
3. Si una caché tiene el bloque en estado compartido, pasa a estado inválido.

**Acierto de escritura en bloque compartido.**

El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado compartido. El controlador quiere acceso exclusivo al bloque, por lo que difunde un paquete de petición de acceso exclusivo al bloque `PtEx(B)`.

El estado del bloque en la caché después de la escritura será modificado[3].

`PtEx(B)` provoca los siguientes efectos:

1. Si la memoria tiene el bloque válido, lo deposita en el bus y pasa a estado inválido.
2. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus y pasa a estado inválido.
3. Si una caché tiene el bloque en estado compartido, pasa a estado inválido.

**Acierto de escritura en bloque modificado.**

El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado modificado. Como el nodo ya tiene acceso exclusivo al bloque, no se genera ningún paquete. El bloque sigue en estado modificado.

**Acierto de lectura.**

El procesador lee (`PrLec(B)`) y el bloque está en caché en estado compartido o modificado. No se genera ningún paquete y el bloque continúa en su mismo estado.

**Reemplazo.**

Fallo en el acceso a otro bloque y la política de reemplazo de la caché selecciona a `B` para hacer sitio al nuevo. Si el bloque reemplazado se encuentra en estado modificado, el controlador de caché difunde un paquete de posescritura en memoria `PtPEsc(B)`.

El estado del bloque en la caché pasa a ser inválido.

`PtPEsc(B)` provoca el siguiente efecto:

1. El bloque se transfiere a memoria principal, pasando el estado del bloque en memoria a válido.

[3]Notemos que el acierto de escritura en un bloque compartido se trata de forma similar al fallo de escritura.

### 1.2.6. Protocolo MESI (*Modified-Exclusive-Shared-Invalid*) de espionaje

El protocolo que estamos a punto de ver también utiliza posescritura como política de actualización de memoria principal y escritura con invalidación como política para mantener la coherencia entre cachés, como cabría esperar.

Con el protocolo MSI, siempre que se escribía en la copia de un bloque en una caché se genera un paquete de petición con acceso exclusivo al bloque, aunque no haya copias en otras cachés que se tengan que invalidar. Esto generaba un tráfico innecesario por la red que degrada las prestaciones de las aplicaciones, especialmente en aplicaciones secuenciales. Es por esto que el protocolo MESI divide el anterior estado compartido en dos (con la filosofía de que si un bloque está en estado compartido en un único nodo no está siendo compartido realmente).

Un bloque en estado *exclusivo* será válido sólo en dicha caché y en memoria principal. Si un bloque está en estado exclusivo en un nodo y el procesador del nodo escribe en el bloque, no se generará paquete para solicitar uso exclusivo del bloque (invalidando copias), ya que ninguna otra caché tiene copias del bloque. Si un bloque está en estado exclusivo, entonces son dos los propietarios del bloque: la memoria principal y una única caché.

Reordenamos ahora el esquema de disponibilidad de bloques según su estado, desarrollando a continuación cada estado en este nuevo protocolo:

- Inválido.
- Compartido.
- Exclusivo.
- Modificado.

#### Estados de un bloque en caché

**Modificado (M).**

Si el bloque en la caché se encuentra en estado modificado, significa que ese es el único nodo del sistema con una copia válida del bloque, mientras que el resto de cachés y memoria tienen una copia desactualizada (no válida).

El controlador de su caché debe proporcionar el bloque si observa al espiar el bus que algún componente lo solicita y debe invalidar su propia copia si algún otro nodo solicita una copia exclusiva del bloque para su modificación.

**Exclusivo (E).**

Si el bloque en caché se encuentra en este estado, significa que es la única caché en el sistema con una copia válida del bloque: el resto de cachés tienen una copia no válida. La memoria tiene también una copia actualizada el bloque.

La caché debe invalidar su copia si observa al espiar el bus que algún nodo solicita una copia exclusiva del bloque para su modificación.

**Compartido (C).**

Supone que el bloque es válido en esta caché, en memoria y en al menos alguna otra caché.

| Estado | Evento | Acciones del controlador | Estado final |
|---|---|---|---|
| | `PrLec(B)` | | E |
| E | `PrEsc(B)` | | M |
| | `PrLec(B)` | | S |
| | `PrLecEx(B)` | | I |
| I | `PrLec(B)` y no hay cachés con copias de `B` | Genera `PtLec(B)` y recibe `RpBloque(B)` | S |

Tabla 1.4: Tabla de nuevas acciones del controlador de caché.

La caché debe invalidar su copia si al espiar el bus algún otro nodo solicita una copia exclusiva del bloque para su modificación.

**Inválido (I).**
Supone que el bloque no está físicamente en lacaché, o si se encuentra, ha sido invalidado por el controlador como consecuencia de la escritura en la copia del bloque situado en otra caché.

En cuanto a los paquetes que usa MESI, son los mismos que MSI.

### Transiciones de estado de un bloque

En la Tabla 1.4 podemos ver las nuevas acciones del controlador de caché de un nodo que provocan los eventos relacionados con un bloque `B`, teniendo en cuenta el estado del bloque en la caché que ahora origina el protocolo MESI frente a MSI (es decir, tener en cuenta la Tabla 1.3 y agregarle las acciones de la Tabla 1.4).

Detallamos las acciones que se cometen tras la disparación de cada evento con detalle.

**Fallo de lectura.**
El procesaor lee (`PrLec(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de lectura del bloque de memoria `PtLec(B)`.

El estado del bloque en la caché será pasará a ser compartido si hay copias del bloque en otras cachés y exclusivo en caso contrario. Se puede añadir una línea `OR` cableada que informe si hay cachés con copias del bloque solicitado. El estado en memoria será válido.

`PtLec(B)` provoca los siguientes efectos:

1. Si el bloque se encuentra en otra caché en estado modificado, su controlador deposita en el bus el paquete `RpBloque(B)` y pasa a estado compartido. La memoria también recoge el bloque del bus, pasando a estado válido (si tiene estados).
2. Si el bloque está compartido en otras cachés, la memoria proporciona el bloque a la caché que la solicita. El bloque sigue en estado compartido en cada una de las cachés.

3. Si el bloque se encuentra en estado exclusivo, pasa a estado compartido. En este caso, la respuesta con el bloque que llega a la caché del nodo solicitante procede de la memoria principal.

Hay implementaciones en las que si el bloque solicitado está disponible en estado válido en alguna caché, en lugar de proporcionar el bloque la memoria lo proporciona la caché. Es necesario aquí añadir hardware que realice esta funcionalidad (que decida qué caché seleccionar). Esta alternativa es interesante en sistemas con memoria físicamente distribuida (que probablemente use protocolos basados en directorios), ya que se puede proporcionar el bloque desde el nodo más cercano al que lo solicita, reduciendo tiempos.

**Fallo de escritura al no estar el bloque en la caché.**

El procesador escribe (`PrEsc(B)`) y el bloque no está en caché (o está en estado inválido). El controlador de caché difunde un paquete de petición de acceso exclusivo al bloque de memoria `PtLecEx(B)`.

El estado del bloque en la caché después de la escritura será modificado.

`PtLecEx(B)` provoca los siguientes efectos:

1. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus (invalidando la respuesta de la memoria) y pasa a estado inválido.
2. Si una caché tiene el bloque en estado exclusivo o compartido, pasa a estado inválido.

**Acierto de escritura en bloque compartido.**

El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado compartido (es decir, está en algún otra caché también). El controlador quiere acceso exclusivo al bloque, por lo que difunde un paquete de petición de acceso exclusivo al bloque `PtEx(B)`.

El estado del bloque en la caché después de la escritura será modificado[4].

`PtEx(B)` provoca los siguientes efectos:

1. Si la memoria tiene el bloque válido, lo deposita en el bus y pasa a estado inválido.
2. Si una caché tiene el bloque en estado modificado, deposita el bloque en el bus y pasa a estado inválido.
3. Si una caché tiene el bloque en estado compartido, pasa a estado inválido.

**Acierto de escritura en bloque modificado o exclusivo.**

El procesador escribe (`PrEsc(B)`) y el bloque está en caché en estado modificado o exclusivo. Como no hay ningún otro nodo con acceso al bloque, no se genera ningún paquete. El bloque pasa a estado modificado si estaba en estado exclusivo y si no, no se modifica su estado.

[4]Notemos que el acierto de escritura en un bloque compartido se trata de forma similar al fallo de escritura.

**Acierto de lectura.**

El procesador lee (`PrLec(B)`) y el bloque está en caché en estado compartido, exclusivo o modificado. No se genera ningún paquete y el bloque continúa en su mismo estado.

**Reemplazo.**

Fallo en el acceso a otro bloque y la política de reemplazo de la caché selecciona a `B` para hacer sitio al nuevo. Si el bloque reemplazado se encuentra en estado modificado, el controlador de caché difunde un paquete de posescritura en memoria `PtPEsc(B)`.

El estado del bloque en la caché pasa a ser inválido.

`PtPEsc(B)` provoca el siguiente efecto:

1. El bloque se transfiere a memoria principal, pasando el estado del bloque en memoria a válido.

### 1.2.7. Protocolos basados en directorios

Los protocolos de espionaje son apropiados para redes que implementan comunicaciones con comunicaciones uno-a-todos, tales como los buses. Sin embargo, para redes en las que las difusiones son costosas o cuando necesitamos una gran escalabilidad en el sistema, es mejor usar protocolos basados en directorios. Por tanto, nos serán útiles tanto en multiprocesadores con memoria físicamente distribuida (los NUMA) como en multiprocesadores con memoria centralizada (los UMA) con red escalable.

Estos protocolos de mantenimiento de coherencia reducen el tráfico en la red enviando selectivamente órdenes sólo a las cachés que disponen de una copia válida del bloque implicado en la operación de memoria. Para que esto sea posible, debe existir (como ya se ha comentado previamente) una tabla de memoria (o directorio de memoria) en la que haya una entrada de directorio asociada a cada bloque de memoria principal, con información de las cachés que tienen copia de dicho bloque; y de toda la información necesaria para tratar las situaciones de incoherencias.

Por ejemplo, en la Tabla 1.5, representamos una tabla con una entrada por cada bloque de memoria y una fila por cada caché en el sistema, junto con sus bits de presencia. La última fila corresponde con el bit de presencia en memoria principal.

| Bloque | $C_0$ | $C_1$ | ... | $C_{N-1}$ | |
|---|---|---|---|---|---|
| 0 | 1 | 0 | ... | 1 | 0 |
| 1 | 0 | 0 | ... | 1 | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $M-1$ | 1 | 1 | ... | 1 | 0 |

Tabla 1.5: Directorio de memoria, mediante vector de bits.

Según cómo esté distribuido el almacenamiento del directorio en la jerarquía de memoria, podemos clasificar los tipos de directorios de memoria en:

- Directorio centralizado.
- Directorio distribuido entre módulos de memoria principal.
- Directorio distribuido entre módulos de memoria principal y caché.

El primer protocolo basado en directorio que se implementó consistía en un directorio centralizado monolítico, con una entrada para cada uno de los bloques de memoria principal con información de estado e información de las cachés con copias del mismo.

Este directorio tenía que atender todas las peticiones de acceso a memoria generadas por el sistema, lo que introducía un cuello de botella que degradaba las prestaciones del sistema. Buscamos por tanto repartir el directorio en el sistema. Esta repartición puede realizarse tanto por filas como por columnas (o ambas).

En una implementación distribuida entre los módulos de memoria principal, cada módulo tiene asociado un subdirectorio con información para los bloques de memoria de dicho módulo. En este caso, hemos repartido la tabla por filas, de forma que las peticiones podrán ser atendidas por los subdirectorios en paralelo.

En una implementación distribuida entre los módulos de memoria principal y cachés, además de distribuir las filas entre los módulos de memoria principal, se distribuye cada una de las filas entre las cachés con copia del bloque.

### Protocolo MSI para multiprocesadores NUMA

A continuación, vamos a describir dos protocolos MSI con prosescritra e invalidación para NUMA basados en directorios: uno que difunde las peticiones a todos los nodos (con difusión) y uno que envía las peticiones sólo al nodo que tiene el bloque en el trozo de memoria principal más cercano al nodo (sin difusión). Al seguir esta filosofía, nos aparecen tres roles que desempeñan los nodos en la red NUMA:

**Nodo solicitante de un bloque** (S)**.** Es un nodo que genera una petición del bloque (`PtLec`, `PtEx`, `PtLecEx` o `PtPEsc`).

**Nodo origen de un bloque** (O)**.** Recordamos que en un multiprocesador NUMA la memoria está repartida físicamente entre los nodos, de forma que cada uno tiene en módulos de memoria más cercanos un trozo del espacio de direcciones total del multiprocesador. Decimos que esos módulos de memoria más cercanos se hospedan en el nodo. El nodo origen o *home* de un bloque es aquél que tiene el bloque en el trozo de memoria que hospeda.

**Nodo propietario de un bloque** (P)**.** Es un nodo que tiene copia del bloque en su caché.

De esta forma, en la situación más compleja (que estos tres nodos sean distintos), tenemos al controlador de caché de un nodo $S$ que solicita un bloque `B` de memoria principal, por no tenerlo en su caché. Dicha petición la procesará el controlador de caccé del nodo $O$ (origen del bloque), ya que la dirección de `B` corresponde a un módulo de memoria alojado en $O$. Dado que $O$ no tendrá a `B` en estado válido (suponíamos que $O \neq P$), solicitará al nodo $P$ el bloque, que lo tiene en estado válido para así poder devolver a $S$ el bloque solicitado.

| Bloque | Estado Memoria | $C_0$ | $C_1$ | ... | $C_{N-1}$ |
|---|---|---|---|---|---|
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $B-1$ | V | 0 | 0 | ... | 0 |
| $B$ | V | 1 | 1 | ... | 1 |
| $B+1$ | I | 0 | 0 | ... | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Tabla 1.6: Subdirectorio de memoria principal para un nodo de un NUMA.

### 1.2.8. Protocolo MSI para procesadores NUMA sin difusión

Como en este protocolo no usamos difusión, necesitamos almacenar en el directorio de memoria prncipal, además del estado del bloque en memoria, información sobre las cachés con copia del bloque, para que las invalidaciones se puedan propagar sólo a los nodos con copia del bloque.

Al igual que en la Tabla 1.5, se puede usar un vector de bits de presencia para almacenar dicha información. Habría un bit para cada caché conectada a la red NUMA, de forma que un bit activo (un 1) significa que hay una copia válida del bloque en esa caché. El directorio a usar se encuentra distribuido entre módulos de memoria principal (entre los nodos del computador NUMA). En la Tabla 1.6 podemos observar cómo es uno de estos subdirectorios de un nodo, con una entrada por cada bloque de memoria alojado en el nodo. Este protocolo es el más básico a la hora de usar un directorio con posescritura, invalidación y sin difusión.

Todas las peticiones de un bloque se envían al nodo origen del bloque, que es quien tiene el subdirectorio con información sobre el bloque. El nodo origen propagará las invalidaciones a los nodos con copia de un bloque. Con esta propagación, cumplimos la condición 1 de coherencia. El orden de llegada de las peticiones de un bloque al origen será el orden en el que los nodos van a ver los accesos a memoria de ese bloque. Todos van a ver el mismo orden en las escrituras, cumplíendose la segunda condición de coherencia.

#### Estados de un bloque en caché

Son los mismos que teníamos ya en el protocolo MSI de la sección 1.2.5, Modificado (M), Compartido (C) e Inválido (I).

#### Estados de un bloque en memoria

Este protocolo cuenta con dos estados estables que ya tenía el protocolo MSI de espionaje: válido e inválido. Sin embargo, nos encontramos además implementaciones con estados "pendientes" de un estado estable, como pueden ser *pendiente de válido* o *pendiente de inválido*.

Un estado pendiente indica que se está procesando un acceso a memoria del bloque. Hasta que no se pueda procesar una nueva petición sobre dicho bloque, su estado continuará en pendiente. A las peticiones que se reciben de un bloque pendiente se responde con un paquete de reconocimiento negativo al solicitanto, para que intente de nuevo la petición más tarde.

**Válido.** Puede haber copias válidas en una o en más cachés.

**Inválido.** Hay una copia válida en una caché en estado Modificado.

**Pendiete de válido.** Hay una copia válida en una caché, cuyo contenido se trasladará a la memoria próximamente.

**Pendiete de inválido.** La copia del bloque en la memoria está esperando a invalidarse, en cuanto se invalide en el resto de cachés.

### Paquetes generados por controladores e interacciones

A continuación, desarrollamos varias interacciones que pueden suceder entre los tres posibles roles principales de los nodos de un computador NUMA, desarrollando al mismo tiempo todos los paquetes que usan los contorladores de caché de cada nodo:

1. Paquetes de petición (`Pt`) desde $S$ hasta $O$ (cuando son distintos nodos).

   *a*) Cuando se produce una lectura del procesador del nodo $S$ (`PrLec(B)`) con fallo de caché (no hay copia válida del bloque `B` en la caché privada del nodo), se genera el paquete de lectura de bloque `PtLec(B)`.

   Cuando reciba el paquete de respuesta `RpBloque(B)`, introducirá el bloque en la caché, poniendo su estado en Compartido.

   *b*) Cuando se produce una escritura del procesador del nodo $S$ (`PrEsc(B)`) en bloque `B` en estado Compartido, se genera el paquete de petición de acceso exclusivo sin lectura `PtEx(B)`.

   Cuando reciba el paquete de respuesta `RpInv(B)` que confirma la invalidación en memoria y en otras cachés, modificará el bloque y cambiará el estado de `B` a Modificado.

   *c*) Cuando se produce una escritura del procesador del nodo $S$ (`PrEsc(B)`) en el bloque `B` en estado Inválido, se genera el paquete de petición de lectura con acceso exclusivo `PtLecEx(B)`.

   Cuando reciba el paquete de respuesta `RpBloqueInv(B)` que confirma la invalidación en memoria y en otras cachés, junto con el propio bloque `B`, lo modificará y cambiará el estado de `B` a Modificado.

   *d*) Cuando el controlador de caché del nodo $S$ reemplaza un bloque `B` que se encuentra en estado Modificado, se genera el paquete de petición de posescritura `PtPEsc(B)`.

   Posteriormente, el bloque dejará de estar físicamente en la caché.

2. Paquetes de reenvío (`Rv`), de petición desde $O$ a nodos con copia $P$; y paquetes de respuesta (`Rp`), desde $O$ al nodo $S$.

   *a*) Cuando $O$ recibe el paquete `PtLec(B)` y el bloque `B` está en estado Inválido en memoria principal, entonces algún nodo dispone del bloque válido en su caché. Si es el nodo $O$, no se crea paquete de reenvío y se devuelve `RpBloque(B)`; por otra parte, si no es el nodo $O$, se envía el paquete de

reenvío de lectura del bloque `RvLec(B)` a algún nodo que tenga a `B` válido en su caché.

El nodo $O$ pone el bloque en el directorio en estado Pendiente de válido hasta que se actualice el bloque en memoria una vez recibido de la única caché con copia válida. Entonces, pasará a estado Válido y podrá responder a $S$.

*b*) Cuando $O$ recibe el paquete `PtEx(B)` de $S$ y el bloque `B` está válido en cachés de otros nodos ó recibe el paquete `PtLecEx(B)` de $S$ y el bloque `B` está válido tanto en cachés de otros nodos como en memoria, se envía el paquete de invalidación `RvInc(B)` a los nodos que tengan válido el bloque `B`.

El nodo $O$ pondrá el estado de `B` en su directorio a Pendiente de inválido (para que el resto de llamadas no sean atendidas). Cuando $O$ confirma la invalidación en el resto de cachés:

- Dejará activo sólo el bit de presencia de $S$ en el directorio.
- Pondrá el estado de `B` en memoria a Inválido.
- Responderá a $S$ confirmando su invalidación en memoria y en otras cachés con `RpInv(B)` (en caso de `PtLecEx(B)`, incluirá el bloque en el paquete y será de tipo `RpBloqueInv(B)`).

*c*) Cuando $O$ recibe el paquete `PtLecEx(B)` de $S$ y el bloque `B` está Inválido en memoria principal, entonces algún nodo dispone del bloque válido en su caché. Si es el nodo $O$, no se crea paquete de reenvío y se devuelve `RpBloqueInv(B)`; por otra parte, si no es el nodo $O$, se envía el paquete de reenvío de lectura con invalidación, al nodo que tenga válido a `B`.

El nodo $O$ pondrá el estado de `B` a Pendiente de inválido. Cuando $O$ reciba la confirmación de invalidación así como el bloque, pasará a estado Inválido y responderá a $S$ con el bloque confirmando la invalidación, mediante el paquete `RpBloqueInv(B)`.

3. Paquetes de respuesta (`Rp`) desde $P$ hasta $O$.

*a*) Cuando $P$ recibe el paquete `RvLec(B)`, antes de responder cambia el estado del bloque `B` de Modificado a Compartido. Posteriormente, responde con el paquete de respuesta con bloque `RpBloque(B)`.

Cuando $O$ recibe el paquete:

- Escribe el bloque en memoria principal.
- Pasa el estado del bloque de Pendiente de válido a Válido.
- Responde con el bloque al nodo $S$.

*b*) Cuando $P$ recibe el paquete `RvInv(B)`, antes de responder cambia el estado del bloque `B` a Inválido. Posteriormente, responde con el paquete de respuesta confirmando invalidación `RpInv(B)`.

*c*) Cuando $P$ recibe el paquete `RvLecEx(B)`, antes de responder combia el estado del bloque `B` a Inválido. Posteriormente, responde con el paquete de respuesta con bloque confirmando invalidación `RpBloqueInv(B)`.

### 1.2.9. Protocolo MSI para procesadores NUMA con difusión

Al usar difusión, enviamos los paquetes de petición a todas las cachés, tengan o no copia del bloque. Por tanto, no nos es necesario almacenar en el directorio de memoria principal información sobre las cachés con copias del bloque. Se trata del protocolo más sencillo que puede llevarse a cabo con posescritura, invalidación y con difusión.

Los estados de los bloques en caché y en memoria coinciden con los del protocolo MSI para NUMA sin difusión.

#### Paquetes generados por los controladores e interacciones

A continuación, desarrollamos igual que en el caso sin difusión varias interacciones que pueden suceer entre los tres roles principales de los nodos de un computador NUMA, comentando al mismo tiempo los paquetes empleados.

1. Paquetes de petición (`Pt`) desde $S$ hasta cualquier nodo $N$.

   *a*) El nodo solicitante $S$ envía cualquier paquete de petición (son los mismos que se usaban en el protocolo sin difusión) a todos los nodos del computador. Por tanto, lo reciben también el nodo $O$ y los $P$. Son los mismos paquetes que generaba el nodo $S$ al nodo $O$ en el protocolo sin difusión, como consecuencia de los mismos eventos.

2. Paquetes de respuesta (`Rp`) desde nodos $N$ a $O$.

   *a*) Si $P$ recibe el paquete `PtLec(B)` desde $S$ de un bloque `B` que tiene en estado Modificado (estará por tanto en estado Inválido en memoria), $P$ cambia el estado del bloque `B` en su caché a Compartido y genera un paquete de respuesta con bloque `RpBloque(B)`.

   Cuando $O$ reciba el paquete:

   - Escribirá el bloque en memoria principal.
   - Pasará el estado del bloque a válido.
   - Responderá con el bloque al nodo $S$.

   El resto de nodos no hacen nada.

   *b*) Si $P$ recibe el paquete `PtEx(B)` o `PtLecEx(B)` desde $S$ de un bloque `B` que tenga en estado Compartido o Inválido, invalidarán la copia de `B` en sus cachés (en caso de que estuviera en Compartido) y responderán a $O$ confirmando la invalidación con el paquete `RpInv(B)`.

   *c*) Si $P$ recibe el paquete `PtLecEx(B)` desde $S$ de un bloque `B` que tenga en estado Modificado, primero invalidará la copia de su bloque y responderá a $O$ con el bloque, confirmando la invalidación con el paquete `RpBloqueInv(B)`.

3. Paquetes de respuesta (`Rp`) desde $O$ a $S$.

   *a*) Si $O$ recibe del nodo $S$ un paquete `PtLec(B)`:

- Si `B` está en estado válido, genera el paquete de respuesta `RpBloque(B)`.
- Si `B` está en estado inválido, pone el estado del bloque `B` a Pendiente de validación y esperará a que la caché con una copia válida del bloque envíe dicho bloque. En dicho caso, cambiará el estado a válido y generará el paquete de respuesta `RpBloque(B)`.

*b*) Si $O$ recibe del nodo $S$ un paquete `PtEx(B)`:

- Cambia el estado del bloque `B` a Pendiente de invalidación.
- Espera a recibir la confirmación de invalidación de todos los nodos $N$.
- Una vez recibidas todas las confirmaciones, cambia el estado de `B` a inválido y responde con el paquete `RpInv(B)`.

*c*) Si $O$ recibe del nodo $S$ un paquete `PtLecEx(B)`:

- Si tiene a `B` en estado Válido, lo cambiará a estado Pendiente de inválido y esperará a recibir la respuesta de invalidación en todas las cachés de los nodos $N$. Posteriormente, cambiará a estado Inválido y responderá con el paquete `RpBloqueInv(B)`, junto con el bloque `B`.
- Si tiene a `B` en estado Inválido, esperará a recibir la respuesta de invalidación en todas las cachés de los nodos $N$, junto con la respuesta del nodo $P$, que además contendrá el bloque `B`, el cual se actualizará en memoria y enviará a $S$ en el paquete `RpBloqueInv(B)`.

## 1.3. Consistencia del sistema de memoria

El modelo de programación de memoria compartida ofrece varias ventajas al programador respecto al modelo basado en paso de mensajes: evita (o al menos relaja) la necesidad de distribuir la carga de trabajo (código y datos) entre los distintos procesadores. Esta es la principal razón por la que los multiprocesadores han tenido una amplia aceptación tanto en el mundo comercial como en computación científica (frente a los multicomputadores).

Sin embargo, para programar de forma correcta un multiprocesador, los programadores deben conocer bien cómo se comporta el sistema de memoria frente a las instrucciones que el programador coloca en el código paralelo a ejecutar.

### 1.3.1. Objetivos

Esta sección está orientada a adquirir los conocimientos necesarios para:

- Explicar el concepto de consistencia.
- Distinguir entre coherencia y consistencia.
- Distinguir entre el modelo de consistencia secuencial y los modelos relajados.
- Distinguir entre los diferentes modelos de consistencia relajados.

### 1.3.2. Concepto de consistencia

Para ejecutar instrucciones en el procesador es necesario (a veces) disponer de los datos requeridos de la memoria para completar dichas instrucciones. Por tanto, cuanto antes se consiga leer de memoria, antes se completarán estas instrucciones en la CPU. Es por esto que podemos desear que, por ejemplo, las lecturas adelanten a las escrituras que no son tan necesarias en dicho momento (esto es, que lecturas que suceden ciertas escrituras se ejecuten antes que dichas escrituras), con el fin de aumentar las prestaciones.

Notemos que mientras que las lecturas son un simple acceso a caché (de un ciclo) o una serie de envío de paquetes para traer el bloque a caché (en caso de fallo de caché) según lo visto en el apartado de coherencia de memoria, las escrituras siempre provocan el envío de múltiples paquetes, lo que implica un retardo mucho mayor al de las lecturas en promedio.

Podríamos también plantear un modelo en el que lecturas adelanten a lecturas (a aquellas que den fallo en caché y por tanto deben esperar a que la caché obtenga la información, introduciendo un retardo), escrituras que adelanten a lecturas o escrituras que adelanten a otras, obteniendo en cada tipo una leve optimización del código. De hecho, un código que no implemente ningún tipo de adelantamiento, probablemente se ejecute de forma anormalmente lenta. Las herramientas de programación, los compiladores y el propio hardware pueden realizar adelantamientos de instrucciones de acceso a memoria, con el fin de mejorar las prestaciones y reducir

tiempos de forma considerable.

Este tipo de adelantamientos entre instrucciones de acceso a memoria (tanto de lectura como de escritura) sólo se pueden realizar si los accesos a memoria son sobre distintas posiciones de memoria (ya que si no nos toparíamos con dependencias de datos, como de tipo RAW).

Antes de continuar con la sección, introducimos una notación que usaremos a lo largo de esta:

**Notación.** Cuando notemos $A \to B$, estaremos diciendo que la instrucción $A$ precede a la instrucción $B$, pero que por un cierto motivo se ejecutó la instrucción $B$ antes que la $A$. Podemos leer $A \to B$ como "$B$ adelanta a $A$".
Usualmente usaremos esta notación para indicar que, por ejemplo, las lecturas adelantan a las escrituras: $W \to R$.

Motivada ya la razón por la que tenemos que desear que ciertas instrucciones adelanten a otras, podemos ya dar una definición informal de modelo de consistencia de memoria. Un modelo de consistencia de memoria especifica la forma en la que ciertas instrucciones relativas a memoria adelantarán a otras. Sin embargo, este modelo debe garantizar que se siga una secuencia lógica en la forma de los adelantamientos. En concreto, debe garantizarse que:

- Cada lectura de una dirección proporcione el último valor escrito en dicha dirección.
- Si se escribe varias veces en la misma dirección, se debe retornar el último valor escrito (el de la última escritura según el orden del programa).
- Si se escribe en una dirección a la que previamente se ha accedido para leer, no se debe obtener en la lectura previa el valor escrito posteriormente, sino el que había antes de la escritura.
- No se puede escribir en una dirección si la escritura depende de una condición que no se cumple. Esto se ilustra en la Figura 1.3.2, ya que si permitimos adelantamientos, puede que la evaluación de la variable booleana no sea el esperado en el momento de su evaluación, sucediendo la escritura a pesar del valor de la variable.

```
if(var_booleana){
    A = 1;
}
```

Antes de continuar, cabe mencionar que las arquitecturas de memoria para multiprocesadores aprovechan el hecho de que los distintos procesadores pueden emitir instrucciones de acceso a memoria en paralelo, para realizarlas en paralelo. Sin embargo, los procesadores observan estos accesos de forma secuencial. Esto es, en caso de haberse realizado dos accesos en paralelo, los procesadores habrán observado que uno se ha realizado antes que otro.

### 1.3.3. Modelo de consistencia secuencial

Un multiprocesador presenta un modelo de consistencia secuencial si el resultado de cualquier ejecución coincide con el que se esperaría el programador. Definido formalmente, el modelo de consistencia secuencial garantiza:

**Orden del programa.** En cada procesador, se respeta el orden entre las instrucciones de memoria que se indican en el código que cada procesador ejecuta. De esta forma, no se realizan adelantamientos.

**Orden global o Atomicidad.** Todos los procesadores deben ver el mismo orden en el acceso a memoria. De esta forma, ningún procesador puede ver una escritura en memoria antes que el resto de procesadores, por ejemplo.

Una ejecución atómica en los accesos a memoria garantiza el orden global, ya que si una instrucción es atómica, globalmente parece que se ejecuta y completa antes de que comience la siguiente, al no poder introducir ningún otra instrucción durante la ejecución de esta. Por ello que al orden global a veces se le llame "atomicidad".

El modelo de consistencia secuencial es el modelo de consistencia de memoria más simple que podemos considerar.

### 1.3.4. Modelos de consistencia relajados

Antes de definir el modelo de consistencia secuencial, hablábamos de lo provechoso que es que ciertas instrucciones de memoria adelanten a otras, lo que nos permite eliminar latencias y aumentar tiempos de ejecución. Sin embargo, vemos cómo el modelo de consistencia secuencial no permite realizar adelantamiento de instrucciones. Es por esto por lo que se desarrollan los modelos de consistencia relajados, los cuales incumplen de alguna forma el orden del programa, o el orden global, o ambos.

Por ello, cuando queramos definir un modelo de consistencia que seguirá un multicomputador, bastará con especificar qué parte del orden del programa no se cumple (como permitir que lecturas adelanten a escrituras) o qué parte del orden global no se cumple.

Todavía no hemos dado un ejemplo ventajoso que sea resultado de no cumplir el orden global. Procedemos a desarrollarlo: puede ser deseable que, en un mismo procesador, una lectura de una posición de memoria $k$ adelante a una escritura de la misma posición de memoria $k$. Si permitiéramos esto cumpliendo con el orden global, se produciría una clarísima falta de consistencia en el programa (obtenemos una lectura que no coincide con una escritura anterior). Es por esto por lo que puede relajarse el orden global (no cumplirse), de forma que la lectura en vez de acceder a memoria, lea el contenido de la petición de escritura (y por tanto, pudiendo acceder al contenido de la posición $k$ antes que el resto de procesadores).

Notemos que este ejemplo también relaja el orden del programa (al permitir que lecturas adelanten a escrituras).

Actualmente, todos los modelos de consistencia de memoria permiten, al menos, el adelantamiento $W \rightarrow R$ (lecturas adelantan escrituras), debido al retardo que eliminan en los programas. El modelo de consistencia secuencial sólo tiene utilidad teórica, al producir programas con grandes latencias en la práctica.

### 1.3.5. Consistencia software y hardware

Todo este tiempo hemos estado hablando de consistencia de memoria a nivel hardware, que es la consistencia que implementa el propio hardware. Aquel programador que programe a bajo nivel (como en ensamblador o que se encuentre desarrollando un sistema operativo) debe conocer bien el sistema de consistencia de su hardware, con el fin de desarrollar códigos que hagan lo que se espera que hagan y no se obtengan errores indeseados difíciles de localizar.

Sin embargo, nos encontramos con que herramientas de programación y compiladores pueden realzar adelantamientos de instrucciones de memoria con la finalidad de optimizar el código e incrementar las prestaciones de nuestro programa. Estas utilidades software siguen unas filosofías de reordenación (cada una la que desee) a las que nos referiremos como modelos de consistencias software.

De esta forma, los modelos de consistencia software proporicionan al programador una nueva capa de abstracción a la hora de desarrollar código a alto nivel. El programador no tiene por qué conocer el modelo de consistencia hardware de su máquina. Serán las propias herramientas de programación las que determinen cómo reordenar el código para que todo parezca haberse ejecutado siguiendo un modelo de consistencia secuencial (aunque este nunca se aplique en realidad).

### 1.3.6. Tipos de modelos de consistencia relajados

Como hemos comentado ya, en realidad los modelos de consistencia relajados son en realidad los verdaderos modelos de consistencia útiles (ya que si no, trabajaríamos con el modelo de consistencia secuencial, que deja ejecutar programas con latencias enormes). Para describir dichos modelos nos es suficiente con describir qué relaja:

- Indicar por qué no se garantiza el orden del programa: qué órdenes entre instrucciones no se garantizan debido a adelantamientos.
- Indicar por qué no se garantiza el orden global: en qué casos un procesador puede acceder de forma prematura a una posición de memoria.
- La necesidad de incluir nuevas instrucciones máquina en el repertorio del procesador que nos permitan mantener el orden secuencial.

La tercera cuestión no la hemos mencionado hasta el momento y es de crucial importancia: si disponemos de un modelo de consistencia que no nos permite especificar que un cierto código debe ejecutarse de forma secuencial sin permitir ningún adelantamiento, no tenemos forma de desarrollar códigos que implementen sincronizaciones entre procesadores (ya que estos códigos necesitan actualizar direcciones de memoria

en orden, protocolos que fallarán si se reordenan las instrucciones de memoria).

Cabe destacar también que la necesidad de disponer de códigos de sincronización es vital. Si no disponemos de códigos de sincronización y nos disponemos, por ejemplo, a ejecutar el siguiente código de forma que la línea 1 sea ejecutada por un procesador y la línea 2 por otro, no tenemos forma de garantizar que la variable `k` del segundo procesador sea igual a 1.

```
1  int A = 1;
2  int k = A;
```

Es por esto por lo que necesitamos tener códigos de sincronización, los cuales para implementarlos necesitan garantizar de alguna forma la secuenciación de las instrucciones de acceso a memoria. De donde surge la necesidad de que los modelos de consistencia de memoria relajados implementen dichas instrucciones.

A continuación, desarrollamos dos modelos de consistencia de memoria relajados distintos, que nos permitirán terminar de entender la importancia de la consistencia.

### Modelo de consistencia de los procesadores x86

Es el modelo de consistencia relajado más sencillo. Para describirlo, desarrollamos las tres cuestiones que comentamos anteriormente:

- Orden del programa: Permite que las lecturas adelanten a las escrituras: $W \rightarrow R$.
- Orden global: Sólo permite acceso adelantado a sus propias modificaciones de la memoria.
- Instrucciones para garantizar un orden: `xchange` y la posibilidad de incluir `lock` en las instruccines aritméticas y lógicas.

Ambas instrucciones garantizan un orden (es decir, todas las instrucciones de lectura y escritura que precedan a `xchange` o a una con `lock` se ejecutarán antes que estas y todas las que sucedan se ejecutarán después de estas instrucciones). La instrucción `xchange` implementa una especie de *swap* (intercambio) entre un registro y memoria. Por otra parte, si incluimos `lock` delante de una instruccion aritmética o lógica, nos permite implementar un cerrojo en la ejecución de la instrucción (en realidad, se produce un acceso al bus de forma exclusiva por parte del procesador que lanzó la instrucción con `lock`).

Recordemos que los procesadores x86 tienen una arquitectura de instrucciones CISC (instrucciones complejas), pero que en realidad tienen un procesador RISC de forma interior con un decodificador que cambia instrucciones CISC por RISC. De esta forma, en realidad la instrucción `xchange` realiza una lectura y una escritura, pero de forma atómica.

### Modelo de consistencia de los procesadores ARMv7

A modo de resumen, podemos decir que su modelo de consistencia lo relaja todo:

- Orden del programa: Permite cualquier tipo de adelantamiento.
- Orden global: Lo relaja totalmente.
- Instrucciones para garantizar un orden: `dmb`.

La instrucción `dmb` sólamente tiene la utilidad de garantizar la ejecución de las instrucciones a memoria que se incluyen antes de ella antes de ejecutar la instrucción `dmb`, además de ejecutar las instrucciones que se encuentran tras ella sólamente tras su ejecución.

```
1  int A = 1;
2  int B = A;
3  int C = A + B;
4  dmb;
5  int A = C + B;
6  int C = C - A;
7  int B = B + 1;
```

Por ejemplo, en el pseudocódigo de la Figura 1.3.6 (la instrucción `dmb` es a nivel de ensamblador y el resto de código es código de C), ninguna de las instrucciones de la línea 5 hacia abajo puede adelantar a ninguna de las instrucciones hasta la lína 3; y ninguna de las instrucciones de hasta la línea 3 puede ser ejecutada tras una instrucción que comienza a partir de la línea 5.

Podemos pensar en la instrucción `dmb` como en una especia de barrera (no confundir con una barrera en el contexto de sincronización) relativa a la memoria: cuando llegamos a una instrucción `dmb` se asegura que ninguna de las instrucciones anteriores se haya ejecutado y que todas las siguientes no han comenzado a ejecutarse todavía.

## 1.4. Sincronización

La comunicación entre distintos procesadores cuando estos comparten memoria (como en multiprocesadores), se lleva a cabo a través de variables compartidas. Debemos saber cómo funcionan los protocolos que permiten realizar sincronizaciones, además de implementarlos, todo esto teniendo en cuenta el modelo de consistencia hardware de nuestro computador, que es de vital importancia a la hora de desarrollar códigos que permitan realizar sincronizaciones, los cuales son muy propensos a alterar su funcionalidad por los adelantamientos de instrucciones de memoria.

### 1.4.1. Objetivos

Tras esta sección, debería ser capaz de:

- Explicar por qué es necesaria la sincronización en multiprocesadores.
- Describir las primitivas para sincronización que ofrece el hardware.
- Implementar cerrojos simples, cerrojos con etiqueta y barreras a partir de instrucciones máquina de sincronización y ordenación de accesos a memoria.

### 1.4.2. Necesidad de sincronización

Motivamos la necesidad de la sincronización entre procesadores con el siguiente ejemplo, que probablemente sea el más sencillo que podamos considerar.

**Ejemplo.** Estamos desarrollando un programa paralelo que será ejecutado por dos hilos (que intentaremos que se ejecuten en dos procesadores de forma paralela). En un determinado instant, queremos copiar el contenido de la variable compartida `A` actualizada por el procesador 1 al procesador 2 (la numeración no es relevante). Plasmamos en código secuencial lo que queremos hacer:

```
1  A = valor;  // código a ejecutar en P1
2  copia = A;  // código a ejecutar en P2
```

De esta forma, el programa que tengamos en el procesador 1 tendrá en alguna parte de su código la instrucción de la línea 1 y el procesador 2 tendrá la de la línea 2. El problema es asegurarnos de que el procesador 2 ejecute dicha instrucción tras haber ejecutado el procesador 1 su instrucción, porque si no estaremos copiando en el procesador 2 el antiguo valor de la variable `A`, que probablemente no coincida con el valor de `valor`.

Deseamos buscar un código del siguiente estilo (tras esta sección, sabrá cómo implementarlo y cómo usarlo):

```
// -- Código para el procesador 1 --
A = valor;
// Avisar a procesador 2 de la actualización
```

```
// -- Código para el procesador 2 --
// Esperar a que el procesador 1 avise de la actualización
copia = A;
```

No es difícil imaginar una forma en la que usar código basado en variables compartidas para sincronizar ambos procesadores de forma que se ejecute de forma correcta el código anterior (garantizando que la instrucción `A = valor;` del procesador 1 se ejecuta antes que `copia = A;` del procesador 2). Esto lo vemos en el siguiente ejemplo, resaltando la necesidad de tener en cuenta el modelo de consistencia hardware del computador en el que estemos trabajando:

**Ejemplo.** Proponemos el siguiente código como solución del ejemplo anterior:

```
// -- Código para el procesador 1 --
// A y K son variables compartidas, inicialmente K = 0
A = valor;
K = 1;      // Indica que ya se ha copiado el valor

// -- Código para el procesador 2 --
// A y K son variables compartidas, inicialmente K = 0
while(K == 0){;}    // Esperar a que se actualice K
copia = A;
```

Si ahora pensamos en qué ocurriría en los dos casos de ejecución más sencillos:

**El código del procesador 1 se ejecuta antes.** Entonces, se copia en `A` el valor de `valor` y se pone `K = 1`. Al ejecutar el código del procesador 2, tendríamos `K = 1`, por lo que no se realizaría el bucle y se copiaría el contenido de `A`.

**El código del procesador 2 se ejecuta antes.** Entonces, entraríamos en un aparante bucle infinito, el cual se detendría cuando el procesador 1 ejecute su código, copiando ya el contenido de `valor` en `A` y haciendo que el procesador 2 salga del bucle, tras lo cual copiaría el contenido de `A`.

Si pensamos esto, el código funcionaría perfectamente y no obtendríamos ningún error. Sin embargo, no estamos teniendo en cuenta el modelo de consistencia de memoria de nuestro computador (el razonamiento no está bien del todo). Procedemos a distinguir casos:

**Modelo de consistencia secuencial.** En dicho caso, se produce lo que es lógico esperar según el código desarrollado. En este caso el razonamiento anterior es válido.

**Modelo de consistencia de un procesador x86.** En este caso, las lecturas pueden adelantar a las escrituras. Sin embargo, en el código del procesador 1 sólo tenemos dos escrituras, luego se respeta el orden del programa. En el procesador 2, tenemos muchas lecturas (tantas como iteraciones del bucle) y una escritura. Por el orden en el que están también se respeta el orden del programa, el código funcionaría como se espera.

**Modelo de consistencia de un procesador ARMv7.** En dicho caso, los códigos de ambos procesadores no funcionan como esperamos:

- Procesador 1: La escritura en la variable `K` puede adelantar a la de `A`, y por tanto avisando al procesador 2 (según la lógica del código) de que la variable `A` ya ha sido modificada, cuando esto no es cierto.
- Procesador 2: Al no garantizarse orden global, puede suceder que la modificación de la variable `K` se haga visible antes que la modificación de la variable `A` (incluso si el código del procesador 1 se ejecutó en correspondencia con el orden del programa), modificando la variable `A` antes de lo previsto (todavía no se ha actualizado su contenido).

  Cabe destacar que esta es una casuística de error, pero que con este modelo de consistencia podría pasar cualquier cosa: que una lectura adelante a otra, que la escritura en `copia` adelante a una lectura de `K`, ... hay múltiples situaciones en las que sucedería algo no esperado.

Para garantizar el orden secuencial del programa (crucial a la hora de desarrollar este código de sincronización entre dos procesadores) necesitamos, como ya se vió en la sección de consistencia, usar las instrucciones que nos proporcionan los modelos de consistencia para asegurar ejecuciones secuenciales del mismo.

En este caso, sólo debemos introducir dichas instrucciones en el modelo de consistencia de los ARMv7 (en los ejemplos considerados), que sólo dispone de la instrucción `dmb` (la cual nos es suficiente). El código correcto para este tipo de procesadores lo mostramos en el siguiente pseudocódigo (hablamos de pseudocódigo por estar mezclando instrucciones ensamblador como `dmb` con lenguaje C):

```
// -- Código para el procesador 1 --
// A y K son variables compartidas, inicialmente K = 0
A = valor;
dmb;
K = 1;

// -- Código para el procesador 2 --
// A y K son variables compartidas, inicialmente K = 0
while(K == 0){;}
dmb;
copia = A;
```

De esta forma, obtendríamos el resultado deseado para sincronizar los dos procesadores.

Mostramos ahora un último ejemplo como motivación de los dos protocolos a desarrollar que son tan necesarios para realizar la sincronización entre procesadores.

**Ejemplo.** Queremos paralelizar el siguiente código secuencial:

```
1 for(int i = 0; i < n; i++){
2     sum = sum + a[i];
3 }
4 printf(sum);
```

Es decir, queremos paralelizar la suma de un vector y su impresión en consola. Para ello, podríamos pensar en un primer código de la siguiente forma:

```
1  // La variable sump es privada y a, sum son compartidas
2  // inicilmente sum, sump = 0
3  for(i = id_thread; i < n; i = i+num_threads){
4      sump = sump + a[i];
5  }
6
7  sum = sum + sump;
8  if(id_thread == 0) printf(sum);
```

Este presenta un problema que podemos ya intuir: una condición de carrera en la línea 7, al acceder todas hebras a una variable compartida de forma no exclusiva.

Para solucionar la condición de carrera y garantizar el acceso exclusivo a la variable compartida `sum`, podemos hacer uso de un cerrojo (que suponemos ya implementado):

```
1  // La variable sump es privada y a, sum son compartidas
2  // inicilmente sum, sump = 0
3  for(i = id_thread; i < n; i = i+num_threads){
4      sump = sump + a[i];
5  }
6
7  lock();
8  sum = sum + sump;
9  unlock();
10
11 if(id_thread == 0) printf(sum);
```

Hemos ya eliminado la anterior condición de carrera, pero tenemos una en la que nos habíamos fijado, en la línea 11: puede que la hebra 0 llegue al `if` antes de que el resto de hebras hayan terminado de sumar su aportación a la variable `sum`, imprimiendo en pantalla un resultado prematuro (y posiblemente erróneo) de la variable `sum`. Es por ello que en este código tenemos que usar además una barrera (que también suponemos ya implementada):

```
1  // La variable sump es privada y a, sum son compartidas
2  // inicilmente sum, sump = 0
3  for(i = id_thread; i < n; i = i+num_threads){
4      sump = sump + a[i];
5  }
6
7  lock();
8  sum = sum + sump;
9  unlock();
10
11 barrier();
12 if(id_thread == 0) printf(sum);
```

De esta forma, todas las hebras esperarán en la línea 12 antes de proseguir con su código (incluida la hebra 0, que era la que provocaba el fallo a la hora de imprimir).

Hemos visto ya en este ejemplo la necesidad de disponer de cerrojos y barreras. De hecho, con ambas nos es suficiente para desarrollar cualquier código paralelo. Esta sección está dedicada a comentar los tipos de cerrojos y barreras, así como de su implementación.

### 1.4.3. Soporte hardware

A la hora de usar cerrojos o barreras, disponemos de tres opciones:

- Usar las implementadas por las herramientas de programación que estemos usando (en caso de que estas disopngan de las mismas).
- Usar las ya implementadas en el sistema operativo en el que esté trabajando.
- Implementar usted mismo los propios cerrojos y barreras.

Independientemente de la opción que elija, es bueno conocer cómo se implementan estas dos. Antes de proceder a la implementación como tal, necesitamos conocer qué soporte nos ofrece el hardware para su desarrollo.

Como ya se ha comentado, las funciones de sincronización (cerrojos y barreras) utilizan instrucciones que provee el procesador para fijar explícitamente un orden en los accesos a memoria, con las instrucciones que nos provee cada modelo de consistencia de memoria.

Los primeros algoritmos usaban sólo instrucciones de acceso a memoria atómicas (eran una solución, pero de una alta complejidad y grandes tiempos de ejecución al necesitar muchas variables compartidas y operaciones con la finalidad de coordinar a ccesos a secciones críticas). Surgió la necesidad de desarrollar instrucciones que realizasen operaciones en secuencia sobre una misma variable, operaciones de lectura, modificación y escritura (llamadas *lectura-modificación-escritura*[5]) de forma atómica.

Estas instrucciones permiten realizar una coordinación entre procesos de forma rápida y sencilla. Además, el acceso atómico garantiza que ningún otro procesador pueda acceder a la posición durante el acceso.

A continuación, destacamos unas cuantas primitivas que pueden utilizarse para la lectura-modificación-escritura atómica de algunos procesadores (y multiprocesadores).

**Test&Set.**

Realiza de forma atómica sobre una variable `x` las siguientes operaciones:

- Lee en una variable local (es decir, en un registro del procesador) el contenido de la variable `x` y lo devuelve como resultado.
- Escribe un 1 en `x`.

[5]Dudo que este paréntesis sea necesario, pero lo incluyo por si acaso.

**Intercambio.**
Realiza un intercambio de contenidos entre una variable compartida y un registro. Supone leer y escribir en memoria.

En una primitiva de intercambio para sincronización, se realizan (la lectura y escritura) de forma atómica. Es más general que `Test\&Set`, ya que permite almacenar cualquier valor. Un ejemplo es la instrucción `xchange` de los procesadores x86.

**Fetch&Operation.**
Es una familia de primitivas, formada por `Fetch&Add`, `Fetch&Or` y `Fetch&Increment`. Una primitiva de este grupo realiza sobre una variabe `x` las siguientes opreaciones:

- Lee en una variable local (en un registro del procesador) el valor de `x` y lo devuelve como resultado.
- Escribe en `x` el resultado de realizar una operación (que siempre es de naturaleza asociativa y conmutativa) con el contenido de `x` y una variable local `a`.

  Con `Fetch&Add` la operación es una suma, con `Fetch&Or` un OR lógico y con `Fetch&Increment` incrementa en uno la variable `x`. Un ejemplo de `Fetch&Add` es la instrucción `lock xadd reg, mem` ($reg \leftarrow mem$, $mem \leftarrow reg + mem$), de los procesadores x86.

**Compare&Swap.**
Esta opera con dos variable locales `a` y `b` y con una variable compartida `x`. Realiza sobre la variable `x` de forma inninterrumpida las siguientes operaciones:

- Lee el contenido de `x`.
- Si el contenido de `x` coincide con el de `a` se intercambian los contenidos de `x` y de `b`.

**LL y SC.**
Llamado carga enlazada con almacenamiento condicional (*load linked/locked with store conditional*).

ESTOS APUNTES ESTÁN AÚN EN DESAROLLO.