Arquitectura de Computadores

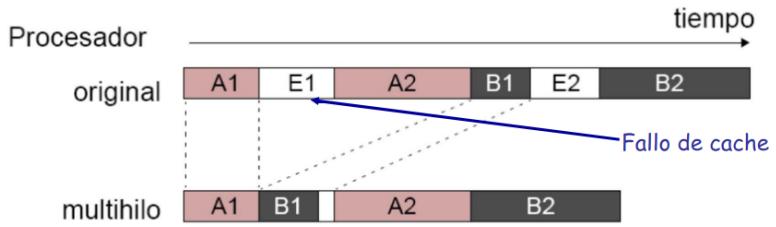
Parte 3

Carmen Azorín Martí

Lección 7: Arquitecturas TLP

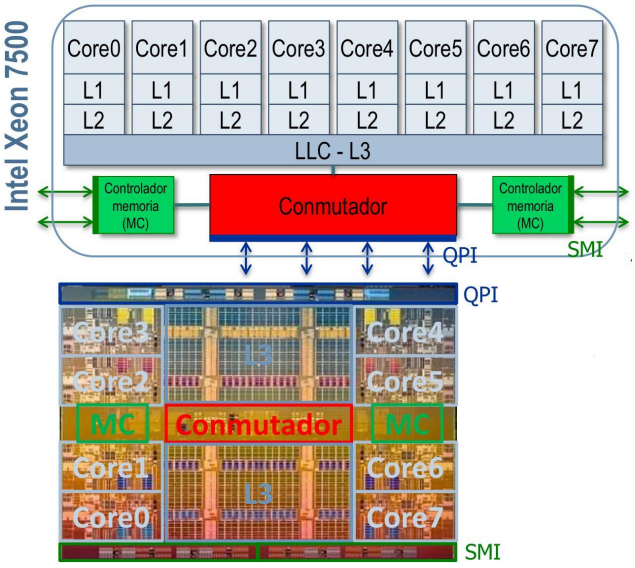
Distinguir entre cores multithread, multicores y multiprocesadores

- **Core multithread**: core que modifica su arquitectura ILP (*instruction level parallelism*) para ejecutar flujos de instrucciones (threads) concurrentemente o en paralelo
    - Es decir, múltiples threads **comparten los recursos del procesador**



    - Incrementa el trabajo procesado por unidad de tiempo
    - El procesador tiene información de todos los threads y con esa información **se van conmutando los flujos de instrucciones** entrelazando su ejecución
    - Ventaja: oculta los tiempos de latencia, ya que cuando un thread está bloqueado, los otros utilizan los recursos
    - Desventaja: retarda la ejecución de cada thread individual

- **Multicore**: ejecutan varios flujos de instrucciones (threads) en paralelo **en un chip** de procesamiento multicore (cada thread en un core distinto)



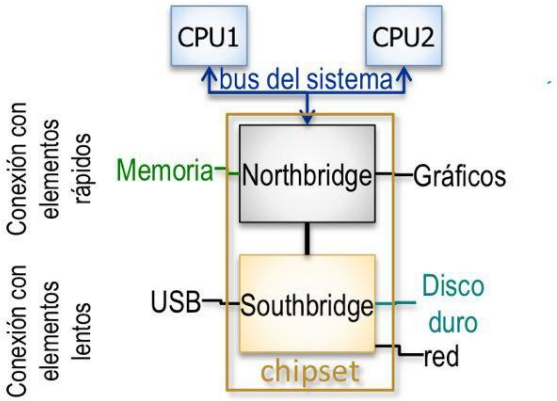
Como se ve en la imagen, comparten la caché de nivel 3 (UMA). Es decir, se tiene una **configuración UMA con respecto a la caché de nivel 3.**

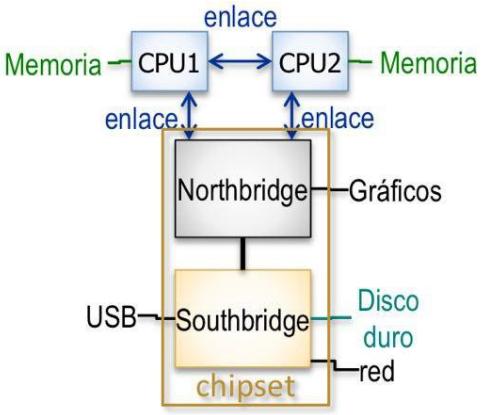
- **Multiprocesador**: ejecutan varios flujos de instrucciones (threads) en paralelo **en un computador** con varios procesadores (cada thread en un core distinto)
    - La **clasificación** de multiprocesadores **según el nivel de empaquetamiento**:
        1. **Chip** (dado de silicio/multicore)

2. **Placa**: tiene zócalos y cada zócalo sirve para conectar un chip con la placa. Por tanto, una placa es una colección de chips (multicores)
3. **Armario**: colección de placas
4. **Sistema**: colección de armarios

- **Clasificación** de multiprocesadores **según el sistema de memoria**:

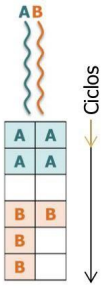
UMA (+ antiguos)	Mayor latencia	Poco escalable	Controlador de memoria en chipset (Northbridge)	Red: bus (medio compartido)
NUMA (+ recientes)	Menor latencia	Escalable	Controlador de memoria en chip del procesador	Red: enlaces y conmutadores





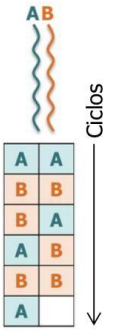
Comparar cores multithread de grano fino, grano grueso y simultánea

- Temporal multithreading (**TMT**): ejecutan varios threads **concurrentemente** en el mismo core
  - La conmutación (intercambio) entre threads controlada por **hardware**
  - Emite instrucciones de un único thread en un ciclo



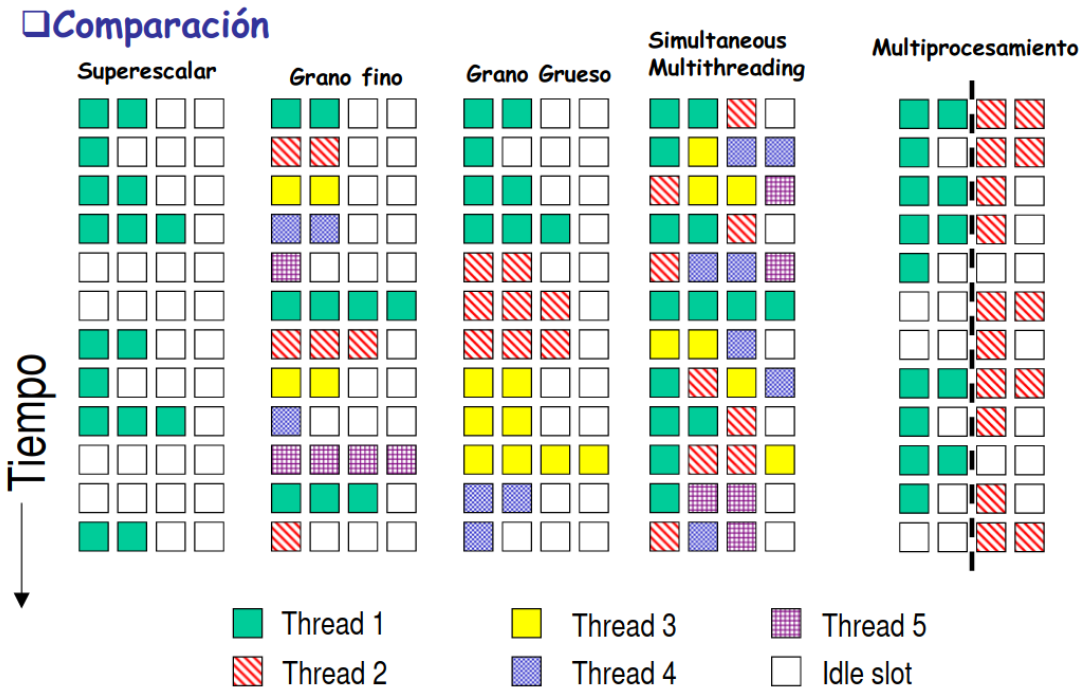
En la imagen vemos que en cada ciclo, sólo se emiten instrucciones de un único thread. (A y B son los threads/hilos/flujos de instrucciones)

- Multithread simultáneo (**SMT**): ejecutan en un core superescalar varios threads **en paralelo**
  - **No** implementa **conmutación** entre threads
  - Pueden emitir instrucciones de varios threads en un ciclo



En la imagen vemos que en cada ciclo, se pueden emitir instrucciones de varios threads.

- Multithread de grano fino (**FGMT**)
    - Se conmuta para saltar los threads bloqueados (round-robin) o por eventos de cierta latencia con técnica de planificación (por ejemplo, thread menos reciente ejecutado)
    - La conmutación entre threads la decide el **hardware cada ciclo** (coste 0)
- Multithread de grano grueso (**CGMT**)
    - Se conmuta entre threads tras intervalos de tiempo prefijados o por eventos de larga latencia (por ejemplo, fallo de caché nivel 2)
    - La conmutación entre threads la decide el **hardware** (coste de 0 a **varios ciclos**)

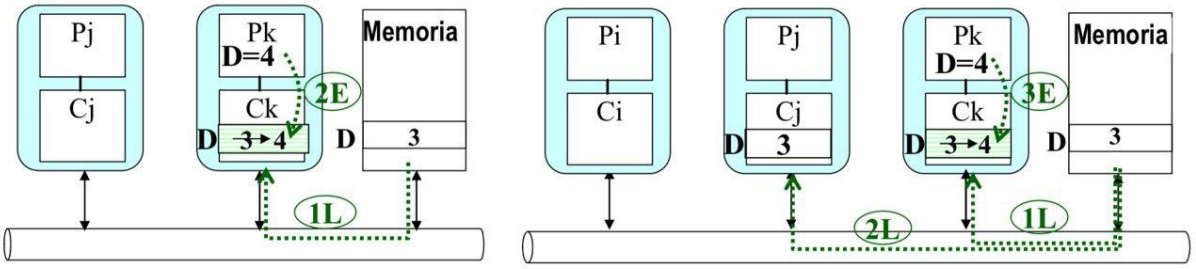


Lección 8: Coherencia del sistema de memoria

Un sistema de memoria incluye cachés, memoria principal, controladores, buffer y redes de interconexión.
Un sistema de memoria tiene copias de bloques de memoria (o direcciones de memoria). De hecho, un sistema de memoria puede tener varias copias del mismo bloque.
Si en el sistema de memoria las copias de una dirección no tienen el mismo contenido, tendremos incoherencia en el sistema de memoria. Y, como es el sistema de memoria el encargado de la comunicación de datos entre procesadores, esto puede provocar problemas.

- Incoherencias en el sistema de memoria:

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



Describir las partes en las que se puede dividir el análisis o el diseño de protocolos de coherencia
Clasificación de protocolos para mantener coherencia en el sistema de memoria:

1. Protocolos de espionaje (**snoopy**)
   Para buses, y en sistemas con una difusión eficiente (pocos nodos o red implementa difusión)
2. Protocolos basados en **directorios**
   Redes sin difusión o escalables (multietapa y estática)
3. Esquemas **jerárquicos**: mezcla de los dos anteriores
   Redes jerárquicas (jerarquías de buses, de redes escalables o de redes escalables-buses)

Diseño lógico en protocolos para coherencia:

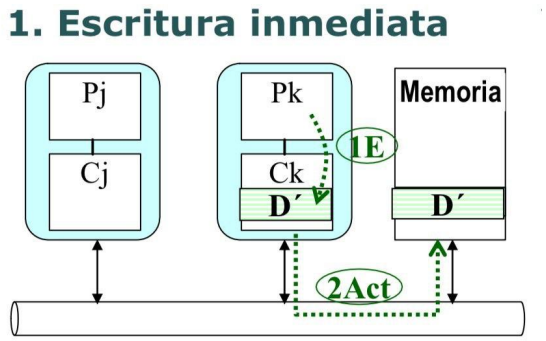
1. Elección tipo **actualización de MP**: escritura inmediata, posescritura, mixta
2. Tipo **coherencia entre cachés**: escritura con invalidación, con actualización, mixta

3. Se debe describir el **comportamiento** del protocolo:
    a. Definir los posibles estados de los bloques en caché y en memoria
    b. Definir transferencias (nodos que intervienen y orden)
    c. Definir transiciones de estados para un bloque en caché y en memoria

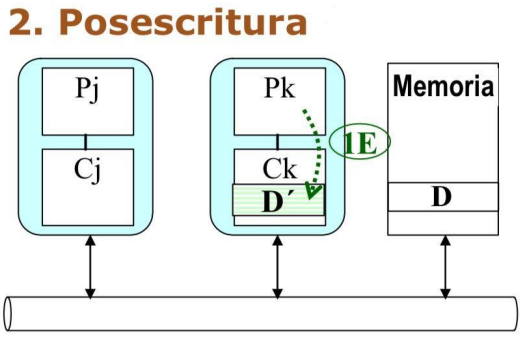
Comparar los métodos de actualización de memoria principal implementados en caché

Hay dos métodos de **actualización de memoria principal** utilizados en caché: escritura inmediata y posescritura.

- **Escritura inmediata**: siempre que se modifica una dirección en caché de un procesador, se modifica en memoria principal.
  No evita incoherencias entre cachés, ni incoherencias caché-MP cuando es la MP la que modifica.
  Por los principios de localidad temporal y espacial, sería más rentable si se escribe todo el bloque en MP una vez realizadas las múltiples escrituras.



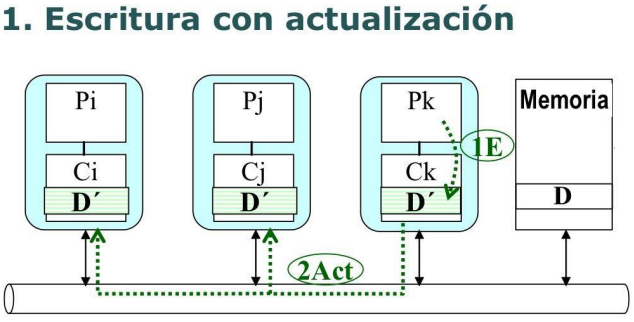
- **Posescritura**: cuando un procesador modifica una dirección sólo se escribe en la caché del procesador.
  La actualización de memoria se realiza posteriormente, cuando el bloque que contiene la dirección modificada se elimina de caché a fin de dejar espacio para otro bloque.
  Se debe, pues, mantener información en el directorio caché sobre los bloques de memoria modificados en la caché.
  No evita incoherencias entre cachés, ni entre caché y MP en ningún caso.



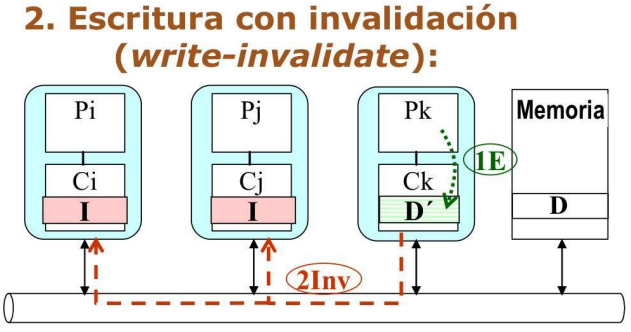
Comparar alternativas para propagar una escritura en protocolos de coherencia de caché

En el apartado anterior, si nos fijamos, no se proponen alternativas que eviten incoherencias entre cachés. Alternativas para propagar una escritura en protocolos de coherencia de caché: escritura con actualización y escritura con invalidación.

- Escritura **con actualización**: siempre que se modifica una dirección en la copia de un bloque de caché, se modifica la dirección en todas las copias del bloque que se encuentren en cachés de otros procesadores.
  Para redes con difusión.



- Escritura **con invalidación**: cuando se va a modificar una dirección en la

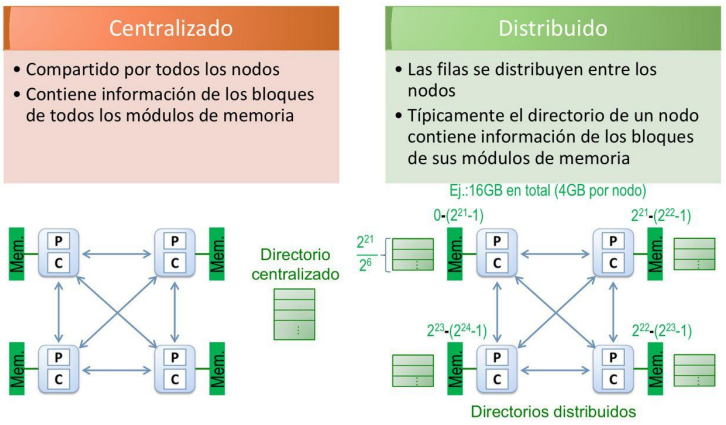


caché de un procesador, primero se invalidan las copias de bloque que contiene esa dirección en otras cachés. Cuando a continuación otro procesador lea la dirección, fallo de caché.
Es más rápido y se utiliza en redes sin difusión.

Distinguir entre protocolos basados en directorios y protocolos de espionaje

Los **protocolos de espionaje** se basan en un bus que hace las transferencias y todos los dispositivos conectados al bus pueden ver (espiar) el contenido del bus. De esta manera, los controladores de caché van actualizando o invalidando las copias de memoria del bloque de su caché.
Dos problemas: en cada transferencia hay que esperar que todas las cachés lean el contenido del bus, y el bus se ocupa de todas las peticiones.
Se utilizan en **difusión simple**.
En los **protocolos de directorios** se mandan los paquetes sólo a las cachés con una copia de ese bloque. Por tanto, se necesita un directorio que contenga información del contenido de las cachés.

1. El directorio puede ser **centralizado**, de forma que la información de los bloques se encuentre fuera de las cachés.
    a. Resulta un cuello de botella (como en el espionaje)
2. Directorio **distribuido** entre los nodos (memorias de las cachés). De esta forma, cada módulo tiene información únicamente de los bloques que contiene el propio módulo.
    a. Puede procesar peticiones en paralelo
    b. Es el más usado y el que vamos a ver en todos los protocolos



El directorio va recibiendo órdenes y las va ejecutando **en serie**. Los procesadores ven las escrituras en el orden en el que el directorio las procesa. Por ello, el directorio debe **garantizar orden**.
Se utilizan en **difusiones complejas**.
Los protocolos de **espionaje no** son **escalables**, ya que a más procesadores no se garantiza más velocidad (porque el bus debe responder más peticiones). En cambio, los protocolos de **directorios sí** son **escalables**, ya que se pueden procesar las peticiones en paralelo.

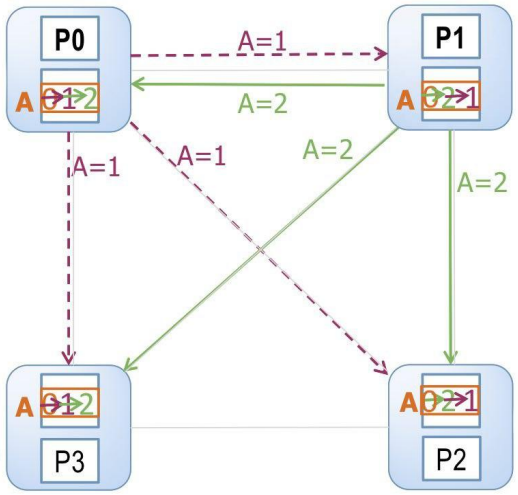
Explicar qué debe garantizar el sistema de memoria para evitar problemas por incoherencias

Se debe tener en cuenta que puede ocurrir que la propagación de una escritura a todos los procesadores a través del sistema de comunicación, requiera un tiempo distinto para cada procesador. Entonces, si se solapan varias propagaciones de escrituras en la misma dirección, distintos procesadores podrían recibir y leer en distinto orden las escrituras en la misma dirección. Debe tenerse en cuenta, por ejemplo, que hay redes de interconexión que no garantizan que los paquetes lleguen al destino en el mismo orden en el que se inyectaron en la red.

Para evitar problemas por incoherencias, dos requisitos:

1. La escritura en una dirección se debe hacer visible en un tiempo finito a otros procesadores (**propagar escrituras**).-> escrituras visibles a todos
    a. Usando difusión se mandan los paquetes a todas las cachés
    b. Para mayor escalabilidad y eficiencia: se mandan los paquetes sólo a la las cachés con copias del bloque
2. El sistema de memoria debe realizar en serie (una después de otra) las operaciones de escritura en la misma dirección, para que así todos los procesadores vean estas escrituras en el mismo orden (**serializar escrituras**).-> en el mismo orden a todos

Ejemplo :

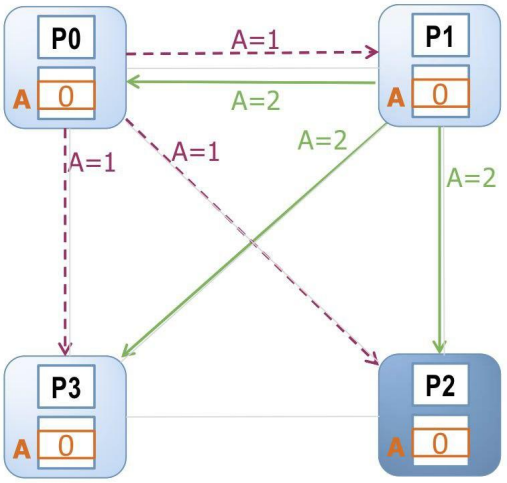


En la figura de arriba se muestra un ejemplo de un sistema de memoria con actualización para **propagar escrituras**. Sin embargo, **no** se asegura la **serialización de las escrituras**. El caso es el siguiente:

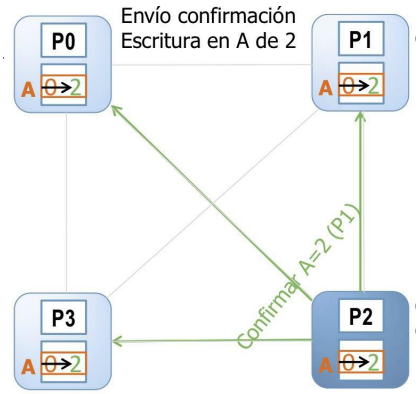
- La caché de P0 pide escribir en A un 1
- La caché de P1 pide escribir en A un 2

En este sistema se utiliza difusión para propagar las escrituras. Sin embargo, no se asegura que se establezca un orden en las escrituras de todas las cachés. Lo que provoca que haya incoherencia. Porque a cada caché le llega la escritura en un orden diferente.

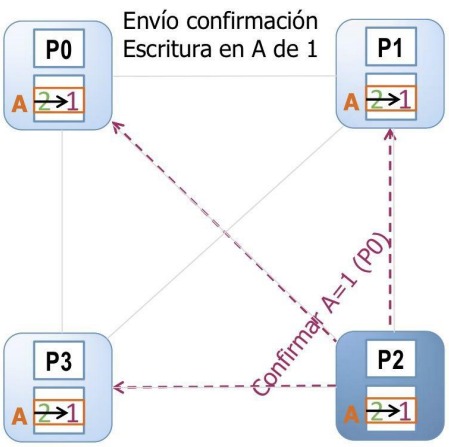
Ejemplo:



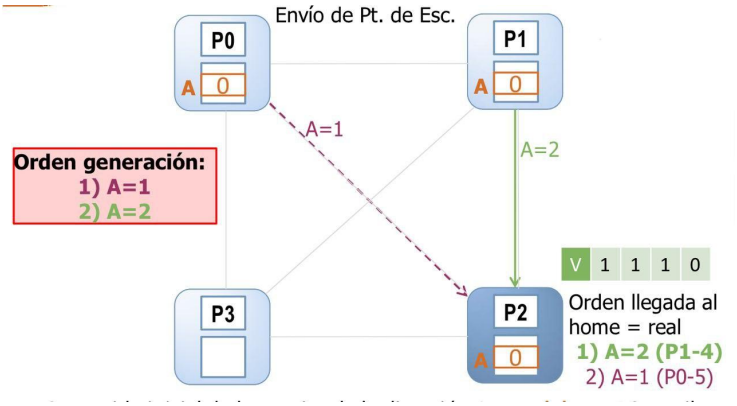
Se quiere hacer el ejemplo anterior pero asegurando la serialización de las escrituras.
Se tiene un directorio distribuido. El bloque de memoria que contiene a A se encuentra en el directorio de P2. Por tanto, P2 es el nodo home que garantizará el orden.
P0 y P1 realizarán la difusión de las respectivas escrituras de A. Estas escrituras sólo serán recibidas por el nodo home (P2). Supongamos que primero recibe A=2 y después A=1.



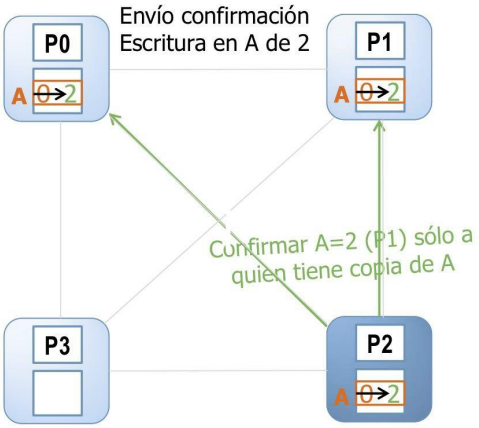
Entonces, el nodo home, en el orden que le han llegado a él, debe confirmar las escrituras al resto de cachés. Así, primero confirma A=2, porque primero le había llegado la orden de escritura de P1. Y, posteriormente, confirma A=1, de P0.



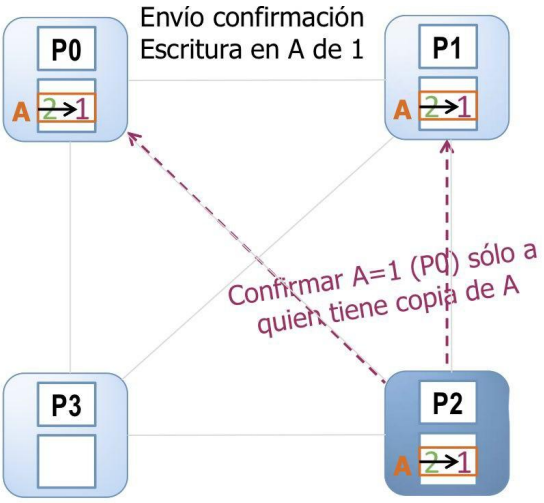
Ejemplo:



Ahora el mismo ejemplo que antes pero sin utilizar difusión (sólo se mandan paquetes a aquellas cachés que tengan copia del bloque).
Como vemos en la imagen, P2 tiene una tabla que le informa de cuáles son las cachés con copia del bloque que contiene la dirección de A. En este caso, son las cachés de P0, P1 y P2.
Se realizan las órdenes comentadas antes (P1 pide A=2 y P0 pide A=1).
Como es sin difusión, P0 y P1 deben pedirle sólo al nodo home (P2) que escriba en A.
P2 recibe las escrituras en un orden (primero A=2 de P1 y luego A=1 de P0).



Luego, es P2 quien debe confirmar las escrituras sólo a aquellas cachés con copia del bloque (P0 y P1). Primero confirma la escritura A=2. Finalmente, confirma la escritura A=1.



Explicar protocolo de mantenimiento de coherencia de espionaje MSI

Como hemos visto, los protocolos de espionaje se basan en la **difusión de paquetes** asociados al mantenimiento de coherencia **a todas las cachés** mediante un bus.

El protocolo MSI se basa en: **posescritura** como política de actualización de memoria y **escritura con invalidación** para la coherencia entre cachés.

Falta entonces definir el comportamiento del protocolo.

En primer lugar, los **estados** de las cachés y la MP:

- Estados de un bloque en caché:
    1. **Modificado** (M): significa que es el único componente en el sistema con una copia válida del bloque; el resto de cachés y memoria tienen una copia no actualizada.
    2. **Compartido** (C): todas las copias del bloque que pueda haber en el sistema están actualizadas.
    3. **Inválido** (I): el bloque no está físicamente en la caché o ha sido invalidad por el controlador de caché como consecuencia de la escritura en la copia del bloque de otra caché.
- Estados de un bloque de memoria:
    1. **Válido**: copia válida
    2. **Inválido**: la copia no es la última actualización (porque otra caché está en estado M)

En segundo lugar, definir las **transferencias** (paquetes):

Una caché, puede generar las siguientes transferencias como consecuencia de acciones de lectura/escritura o de paquetes recibidos:

- Petición de lectura de una bloque (**PtLec**): el procesador de la caché pide leer una dirección de memoria que no se encuentra en su caché (PrLec). El controlador de caché pone en el bus la dirección a la que se desea acceder. El sistema de memoria proporcionará el bloque donde se encuentra la dirección.
- Petición de acceso exclusivo (**PtLecEx**): el procesador de la caché escribe (PrEsc) en una dirección cuyo bloque está C o I. El controlador de caché indica la dirección en la que se desea escribir. El resto de cachés y la MP con copias válidas, invalidan sus copias.
- Petición de posescritura (**PtPEsc**): cuando se reemplaza un bloque Modificado en la caché por otro que no está. Si el bloque está en estado modificado, se debe transferir a MP.
- Respuesta con bloque (**RpBloque**): la caché observa en el bus que otra caché quiere leer o escribir en un bloque que se tiene en estado modificado.

EST. ACT. k	EVENTO	ACCIÓN	SIGUIENTE
Modificado (M)	PrLec/PrEsc		Modificado
	PtLec	Genera paquete respuesta (RpBloque)	Compartido
	PtLecEx	Genera paquete respuesta (RpBloque) Invalida copia local	Inválido
	Reemplazo	*Genera paquete posescritura (PtPEsc)*	*Inválido*
Compart. (S)	PrLec		Compartido
	PrEsc posescr	Genera paquete PtLecEx (PtEx)	Modificado
	PtLec		Compartido
	PtLecEx invalido	Invalida copia local	Inválido
Inválido (I)	PrLec	Genera paquete PtLec	Compartido
	PrEsc	Genera paquete PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

Las letras negras contienen los paquetes que le llegan al controlador de su propio nodo.

Las letras naranjas indican que son paquetes que llegan del exterior (del bus).

Las letras azules indican paquetes que le llegan del procesador de otro nodo.

Las filas sombreadas indican que se provoca cambio de estado.

Explicar protocolo de mantenimiento de coherencia de espionaje MESI

Recordamos que los protocolos de espionaje se basan en la **difusión de paquetes** asociados al mantenimiento de coherencia **a todas las cachés** mediante un bus.

El protocolo MESI se basa en: **posescritura** como política de actualización de memoria y **escritura con invalidación** para la coherencia entre cachés.

En primer lugar, los estados de las cachés y la MP:

- **Estados** de un bloque en caché:
    1. **Modificado** (M): significa que es el único componente en el sistema con una copia válida del bloque; el resto de cachés y memoria tienen una copia no actualizada.
    2. **Exclusivo** (E): significa que es la única caché con copia válida del bloque. La MP también tiene la copia actualizada.
    3. **Compartido** (C): el bloque es válido en esta caché, en la MP y en, **al menos**, otra caché.
    4. **Inválido** (I): el bloque no está físicamente en la caché o ha sido invalidad por el controlador de caché como consecuencia de la escritura en la copia del bloque de otra caché.
- **Estados** de un bloque de memoria:
    1. **Válido**: copia válida en una o varias cachés
    2. **Inválido**: copia válida en una caché

En segundo lugar, definir las **transferencias**: las mismas que en el protocolo MSI

Modificado (M)	PrLec/PrEsc		Modificado
	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 (PtEx)	Modificado
	PtLecEx	Invalida copia local	Inválido
Inválido (I)	PrLec (C=1)	Genera PtLec	Compartido
	~~PrLec (C=0)~~	~~Genera PtLec~~	~~Exclusivo~~
	PrEsc	Genera PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

Lo que cambia es lo que está marcado como “tachado”, la profe no encontraba otra forma de destacarlo.

Cuando el controlador encuentra que la copia del bloque está en estado exclusivo:

- Si se le pide la lectura de su propia copia, no cambia de estado.
- Si se le pide la escritura de su propia copia, cambia a M, ya que ahora es la única con copia actualizada (válida).
- Si otra caché le pide leer una dirección de su copia, ya no será la única con copia actualizada. Ahora, al menos, otra caché tiene copia actualizada (C).
- Si otra caché pide una lectura exclusiva, invalida su copia local, poniendo su estado a I.

Explicar protocolo de mantenimiento de coherencia MSI basado en directorios sin difusión

Recordamos que para redes con difusión más costosa o cuando se necesita mayor escalabilidad, se utilizan **protocolos basados en directorios**.

Cada fila del directorio está asociada a un bloque de memoria y en cada columna, las cachés que tienen copia válida del bloque o alguna otra información.

De nuevo, se utiliza **posescritura** para actualizar la MP y **escritura con invalidación** para asegurar la coherencia entre cachés.

Veamos los **estados** de los bloques en caché y en MP:

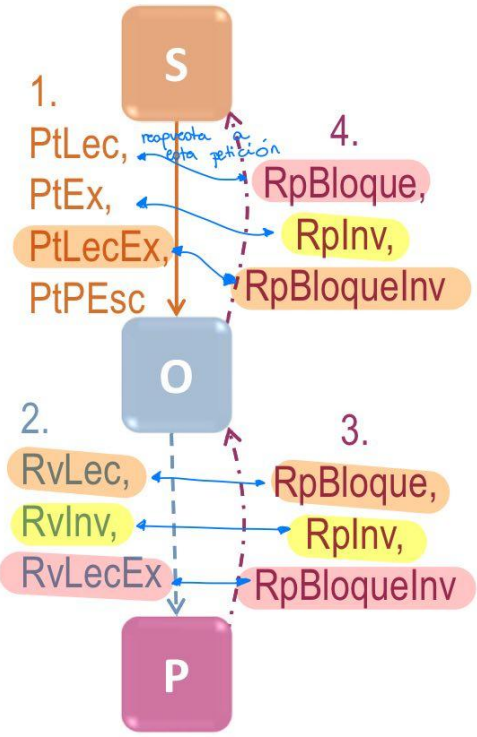
- Estados de un bloque en caché:
    1. **Modificado** (M)
    2. **Compartido** (C)
    3. **Inválido** (I)
- Estados de un bloque de memoria:
    1. **Válido**
    2. **Inválido**

Antes de explicar las transferencias, debemos destacar los **tipos de nodos** que hay:

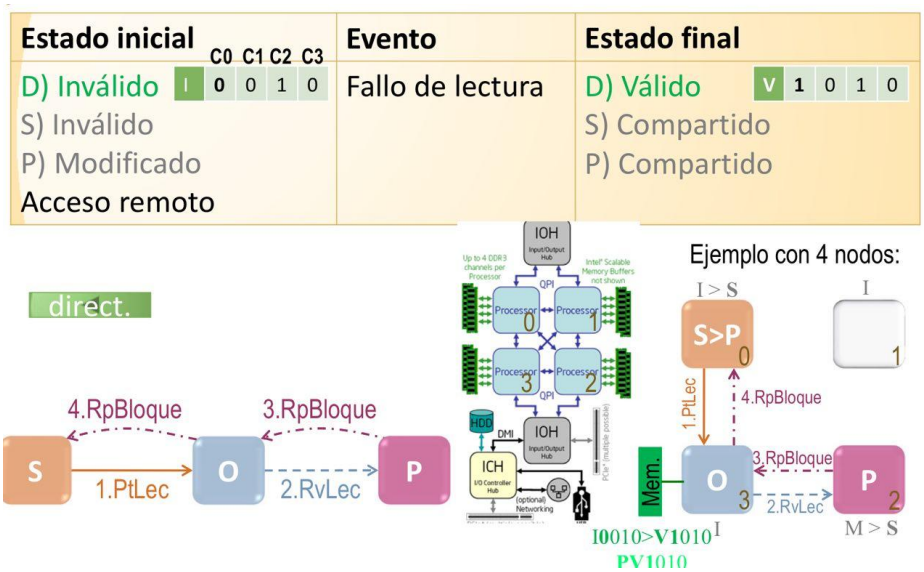
- **Origen** (O) (home): contiene la entrada del directorio para el bloque de memoria (contiene la información sobre qué cachés tienen copia válida del bloque)
- **Solicitante** (S): nodo del procesador que emite una petición sobre el bloque
- **Propietario** (P): caché con copia válida del bloque. Puede estar:
    - **Compartido** (C): es una de las cachés con copia válida del bloque
    - **Modificado** (M): es la única caché con copia válida del bloque de memoria

Las **transferencias** (paquetes) son las siguientes:

- **Petición** (del nodo S al nodo O): puede pedirle
    - Lectura de un bloque (PtLec)
    - Lectura con acceso exclusivo (PtLecEx)
    - Acceso exclusivo sin lectura (PtEx)
    - Posescritura (PtPEsc)
- **Reenvío** (del nodo O al P): el nodo origen como respuesta a las peticiones pide
    - Invalidación (RvInv)
    - Lectura (RvLec, RvLecEx)
- **Respuesta** (del nodo P al O):
    - Respuesta con bloque (RpBloque)
    - Respuesta sin bloque confirmando invalidación (RpInv)
    - Respuesta con bloque confirmando invalidación (RpBloqueInv)
- **Respuesta** (del nodo O al S):
    - Respuesta con bloque (RpBloque)
    - Respuesta sin bloque confirmando invalidación (RpInv)
    - Respuesta con bloque confirmando invalidación (RpBloqueInv)



Ejemplo:



Contexto: MSI (posescritura y escritura con invalidación) con directorios (distribuido) y sin difusión.
Tenemos cuatro nodos y el nodo 0 quiere leer una dirección de memoria (PtLec).
El nodo home (3) tiene la siguiente información sobre el bloque:

- Nodo 0 sin copia válida del bloque que contiene la dirección de memoria (estado de la caché I)
- Nodo 1, sin copia válida (estado de la caché I)
- Nodo 2, con copia válida (la única) (estado de la caché M)
- Nodo 3, sin copia válida (estado de la caché I)

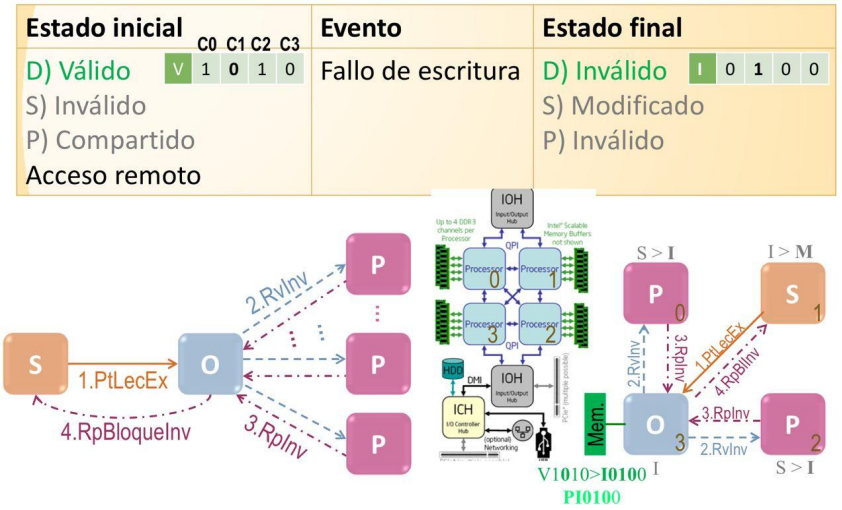
Como solo hay un nodo con copia válida, el estado del bloque de memoria es Inválido (I).
El nodo 0 manda el paquete (PtLec) al nodo home, con la tabla (I-0010).
El nodo home ve que el nodo 2 es el único con copia válida, así que le manda a dicho nodo un paquete de reenvío de la petición de lectura (RvLec). El nodo home se queda con la tabla (PV-1010), PV significa **pendiente de validación**. No puede ponerlo a válido hasta que no reciba la respuesta del nodo 2.
Efectivamente, el nodo 2 le manda como respuesta el bloque pedido (RpBloque).
El nodo home (3) ya puede responder al nodo solicitante (0) con el bloque (RpBloque). Es ahora cuando el nodo home actualiza su tabla a (V-1010). Puede hacerlo porque el nodo 0 ya tiene copia válida del bloque. Ahora serían dos cachés las que tienen copia válida, así que se pasa el estado del bloque en memoria de inválido (I) a válido (V).

Entre toda esta historia, también se puede añadir que el nodo 2 (propietario) pasa de tener el estado del bloque en caché modificado (M) a compartido (C), pues ya no es el único con copia válida del bloque.
El nodo 0 (solicitante) pasa de inválido (I) a tener estado del bloque en la caché compartido (C), pues ya tiene copia válida del bloque.

Cada vez que digo nodo, me refiero a **controlador de caché**.

También hay que comentar que en el tiempo en el que el estado del bloque de memoria es pendiente de validación (PV), si el nodo home recibe otra petición del mismo bloque de memoria, no la procesará todavía. Sino que la deja en espera hasta que no se PV.

Ejemplo:



El contexto es el mismo que en el ejemplo anterior.
Ahora el nodo home (3) tiene la siguiente información:

- Nodo 0, tiene copia válida del bloque donde se encuentra la dirección (estado de la caché C)
- Nodo 1, sin copia válida (estado de la caché I)
- Nodo 2, copia válida (estado de la caché C)
- Nodo 3, sin copia válida (estado de la caché I)

Como hay más de un nodo con copia de la caché válida, el estado del bloque de memoria es válido (V).
El nodo 1 (solicitante) manda un paquete PtLecEx porque quiere acceso exclusivo. El paquete le llega al nodo home (3), que observa que los nodos 0 y 2 tienen copia válida del bloque, ya que en su tabla aparece (V-1010).
Así que, el nodo 0 reenvía la petición como RvInv (para que invaliden sus copias los nodos 0 y 2). Por tanto, se actualiza su tabla a (PI-0100) con PI **pendiente de invalidación**, pues al acabar el ejemplo se espera que solo un nodo (1) tenga copia válida.
Los nodos 0 y 2 responden con RpInv (confirman que han invalidado sus copias). El nodo home (3) manda el paquete de respuesta RpBloqueInv para confirmarle al nodo 1 que ya tiene acceso exclusivo.
El nodo 0 ya puede actualizar su tabla a (I-0100).

Los nodos 0 y 2 (propietarios) han pasado de tener como estado del bloque en la caché compartido (C) a inválido (I).
El nodo 1 (solicitante) ha pasado de tener como estado del bloque en la caché inválido (I) a modificado (M), pues es la única con copia válida del bloque en su caché.

Hay más ejercicios en el Drive y en las diapositivas que la profe recomienda mirar

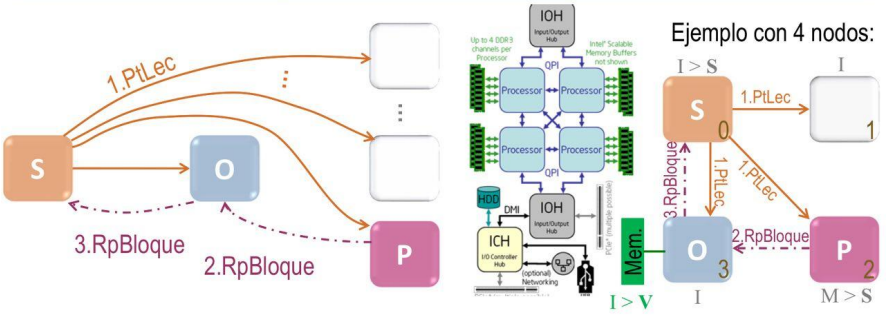
Explicar protocolo de mantenimiento de coherencia MSI basado en directorios con difusión

En este apartado la única diferencia con el apartado anterior son las transferencias:

- **Difusión de petición** (del nodo S al O y P):
  - Lectura de un bloque (PtLec)
  - Lectura con acceso exclusivo (PtLecEx)
  - Acceso exclusivo sin lectura (PtEx)
- **Difusión de petición** (del nodo S al O):
  - Posescritura (PtPEsc)
- **Respuesta** (del nodo P al O):
  - Respuesta con bloque (RpBloque)
  - Respuesta sin bloque confirmando invalidación (RpInv)
  - Respuesta con bloque confirmando invalidación (RpBloqueInv)
- **Respuesta** (del nodo O al S):
  - Respuesta con bloque (RpBloque)
  - Respuesta sin bloque confirmando invalidación (RpInv)
  - Respuesta con bloque confirmano invalidación (RpBloqueInv)

Ejemplo:

Estado inicial	Evento	Estado final
D) Inválido I S) Inválido P) Modificado Acceso remoto	Fallo de lectura	D) Válido V S) Compartido P) Compartido



Contexto: MSI (posescritura y escritura con invalidación) con directorios (distribuido) con difusión.

Como se usa difusión, no se necesita una tabla que diga a qué nodos se deben mandar qué paquetes, porque se van a mandar todos los paquetes a todos los nodos.

El estado inicial del bloque de memoria es inválido (sólo una caché tiene copia válida).

El nodo 0 (solicitante) manda por difusión la petición de lectura de una dirección de memoria (PtLec) a todos los nodos.

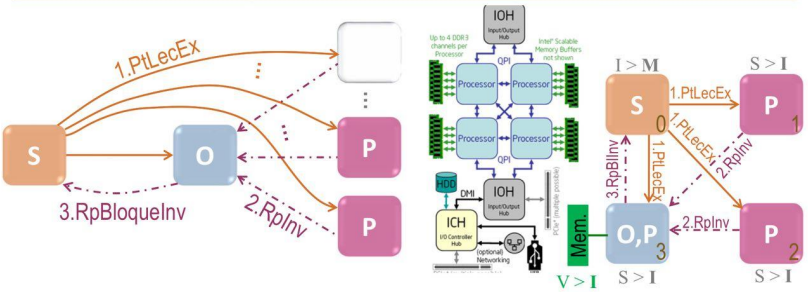
El nodo 2 (propietario) al tener el estado del bloque en la caché a modificado (M), devuelve la respuesta RpBloque al nodo home (3).

El nodo home (3) responde al nodo 0 con RpBloque. Ahora son dos cachés las que tienen una copia válida del bloque. Por tanto, el estado del bloque en memoria se pone a válido (V).

El nodo 0 (solicitante) pasa de tener estado del bloque en la caché inválido (I) a compartido (C). El nodo 2 (propietario) pasa de tenerlo modificado (M) a compartido (C).

Ejemplo:

Estado inicial	Evento	Estado final
D) Válido V S) Inválido P) Compartido Acceso remoto	Fallo de escritura (Procesador escribe)	D) Inválido I S) Modificado P) Inválido

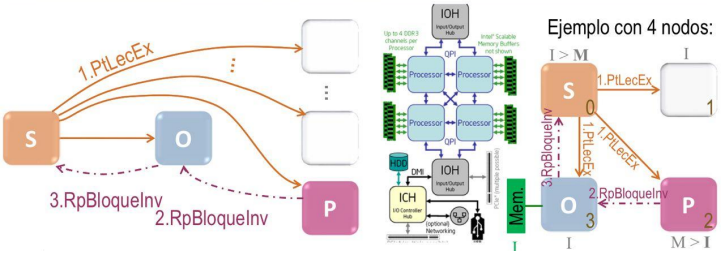


Contexto: igual que el anterior.
Ahora el nodo 0 (solicitante) pide lectura de un bloque de memoria con acceso exclusivo (PtLecEx) por difusión a todos los nodos.
Los nodos 1 y 2 tienen el estado del bloque en la caché a compartido (C). Por tanto, sí tienen una copia válida del bloque y pueden responder.
Los nodos 1 y 2 (propietarios) invalidan sus copias y lo confirman mandándole al nodo home (3) el paquete RpInv.
El nodo 0 cambia el estado del bloque en memoria de válido (porque dos cachés tienen copia válida) a inválido (porque solo una caché tiene copia válida).
El nodo 0 responde al nodo 0 (solicitante) con RpBloqueInv para confirmar que se han invalidado las copias del resto de cachés.

El nodo 0 (solicitante) pasa de tener el estado del bloque en la caché inválido (I) a modificado (M), porque es el único con copia válida.
Los nodos 1 y 2 (propietarios) pasan de tener el estado del bloque en la caché compartido (C) a inválido (I).

Ejemplo:

Estado inicial	Evento	Estado final
D) Inválido I S) Inválido P) Modificado Acceso remoto	Fallo de escritura	D) Válido I S) Modificado P) Inválido



El contexto es el mismo que antes. Ocurre exactamente lo mismo que en el caso anterior. Solamente que hay un nodo que no tiene copia válida del bloque que se pide invalidar. Así que le llega un paquete y no responde a ese paquete.

Lección 9: Consistencia del sistema de memoria

Explicar el concepto de consistencia

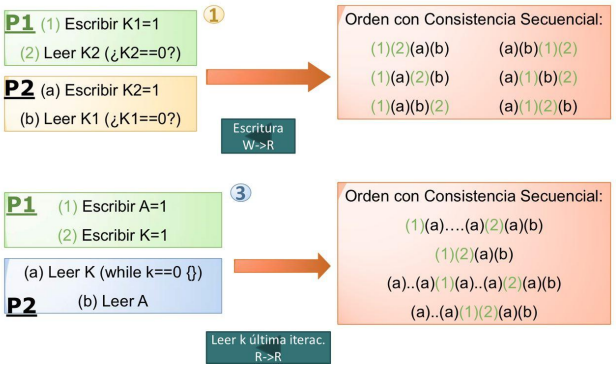
Un modelo de consistencia de memoria especifica el **orden** en el cual las operaciones de acceso a memoria deben **parecer haberse realizado** (operaciones de lectura y escritura).
Es decir, el modelo de consistencia especifica el orden en el que todos los procesadores pueden ver los accesos a memoria de todos ellos.

Distinguir entre consistencia y coherencia

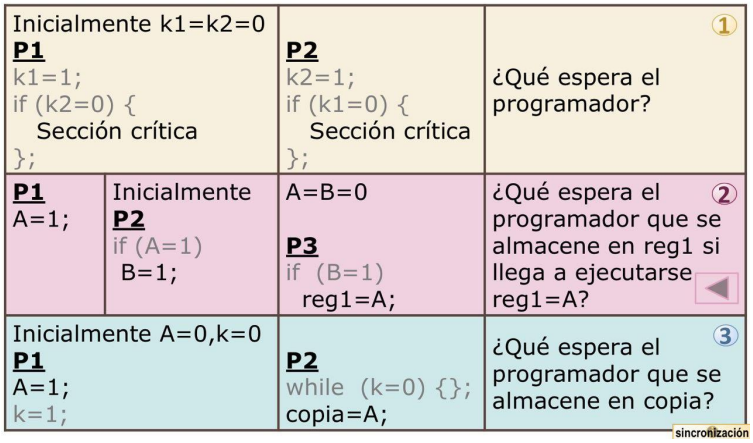
El modelo de **consistencia** de memoria **especifica** las restricciones en el orden en el cual **las operaciones de memoria** deben parecer haberse realizado, incluyendo operaciones **en las mismas o distintas direcciones** (variables) y emitidas por el mismo o distinto procesador.
La coherencia abarca sólo las operaciones realizadas por múltiples componentes **en una misma dirección** (variable), la consistencia abarca todas las operaciones, ya sean o no en la misma dirección.
Por ello, la coherencia está incluida en la consistencia.

Distinguir entre modelo de consistencia secuencial y los modelos relajados

- El modelo de **consistencia secuencial** (SC)
  - El que suele esperar el programador de aplicaciones a **alto nivel**.
  - El resultado de cualquier ejecución coincide con el que se obtendría si las operaciones de todos los procesadores se ejecutaran en algún orden secuencial y las operaciones de cada procesador aparecieran en el orden especificado por su programa.
  - Dos **requisitos**:
    - En cada procesador se debe mantener el orden entre operaciones que indique el código que ejecuta (**orden del programa**)
    - Globalmente se debe mantener un orden secuencial entre todas las operaciones de acceso (**atomicidad**) -> serialización global
  - Este modelo presenta el sistema de memoria a los programadores como si estuviera constituido por una **memoria central global** conectada a los procesadores.
    Dos procesadores no pueden acceder a la memoria central global a la vez.
    Un **conmutador** va conectando y desconectando los procesadores a la memoria para atender las peticiones una después de otra.
  - El modelo de consistencia secuencial no asegura que en cada ejecución del programa se devuelvan los mismos resultados. Por ejemplo:



Ejemplo:



Respondamos a las preguntas.

1) Como nos basamos en un modelo de consistencia secuencial, se espera dos cosas:
   a) Que se mantenga un orden (cada procesador hace su código en orden)
   b) Que mientras un thread lee y escribe en memoria, otro thread no pueda acceder a memoria (atomicidad)

   Por tanto, el programador espera que no se realice ninguna de las secciones críticas (si primero accede a memoria es P1 con k1=1 y el siguiente thread que accede a memoria es el de P2 y escriba k2=1)

   O que se realice una de las secciones críticas (si primero accede P1 poniendo k1=1 y, antes de que acceda P2, vuelve a acceder P1 leyendo el valor de k2=0 o a la inversa).
2) Si se ejecuta reg1=A es porque el thread de P3 al acceder a memoria ha visto B=1. Esto quiere decir que se ha ejecutado B=1; del código de P2. Si se ha ejecutado esa instrucción es porque el thread de P2 en memoria se ha encontrado A=1. Por tanto, antes se había ejecutado el código de P1. Por tanto, reg1=1 seguro.

   Es decir, gracias a que es un modelo de consistencia secuencial, sabemos exactamente el orden en el que se ha accedido a memoria gracias a la atomicidad que asegura.
3) Se espera que almacene 1. La razón es que si se ejecuta copia=A (P2) es porque antes se ha ejecutado el while(k=0) de P2 (por el orden del programa).

   Si ha salido del while es porque k=1, que sólo puede ocurrir si se ha ejecutado la instrucción k=1; de P1. Esta instrucción solo se puede ejecutar si se ha ejecutado antes A=1 (otra vez por el orden del programa).

Con SC, el modelo de programación es simple e intuitivo, pero no permite muchas de las optimizaciones del compilador y del procesador. Por ese motivo, a bajo nivel, el hardware puede ofrecer un modelo que relaja las restricciones sobre el orden de ejecución, con el fin de mejorar las prestaciones.

- **Modelos relajados de consistencia**
  - **Particularidades**:
    - **Orden del programa**: los modelos pueden permitir que en el código ejecutado en un procesador se relaje el orden entre dos accesos a distintas direcciones.
      Permiten alterar el orden entre escrituras (W->W); entre lecturas (R->R); entre una lectura y una escritura posterior (R->W) o la inversa (W->R)
    - **Atomicidad**: hay modelos que permiten que un procesador pueda leer el valor escrito por otro procesador antes de que esta escritura se haga visible al resto de procesadores
  - Una relajación habitual es que un procesador pueda leer con antelación una escritura propia
  - Los modelos relajados comprenden:
    - Especificaciones sobre los órdenes de acceso a memoria que no garantiza el sistema de memoria
    - Mecanismos que ofrece el hardware para forzar de forma explícita un orden cuando sea necesario

Ejemplo:

<table>
<tr><td colspan="2">Inicialmente k1=k2=0
<u>P1</u>
k1=1;
if (k2=0) {
Sección crítica
};</td><td><u>P2</u>
k2=1;
if (k1=0) {
Sección crítica
};</td><td>①
NO se comporta como SC los que relajan el orden W→R</td></tr>
<tr><td><u>P1</u>
A=1;</td><td>Inicialmente
<u>P2</u>
if (A=1)
B=1;</td><td>A=B=0

<u>P3</u>
if (B=1)
reg1=A;</td><td>②
NO se comporta como SC los que no garantizan atomicidad</td></tr>
<tr><td colspan="2">Inicialmente A=0, k=0
<u>P1</u>
A=1;
k=1;</td><td><u>P2</u>
while (k=0) {};
copia=A;</td><td>③
NO se comporta como SC los que relajan el orden W→ W o R → R</td></tr>
</table>

Respondamos a las preguntas:

1) En el modelo SC hemos dicho que se hará 1 sección crítica o ninguna debido al orden del programa y a la atomicidad.
   Si se relaja W->R, entonces P1 podría leer k2 en el if antes de que escribiera k1=1. Por tanto, podría ocurrir que: primero el thread de P2 accede a memoria para leer k1=0 y entra en la sección crítica. Después, el thread de P1 accede a memoria para leer k2=0 y entra en la sección crítica. Posteriormente, P1 realiza k1=1 y, al final, P2 realiza k2=1.
   En este caso, se han accedido a las dos secciones críticas.

2) En el modelo de SC hemos visto que sabríamos el orden en el que los threads de los procesadores han accedido a memoria.
   Si no se garantiza la atomicidad, podría ocurrir que: primero P1 escribe A=1 y, por escritura con actualización, debe escribirse A=1 en el resto de cachés. Pero podría pasar que P2 recibiese la escritura A=1 antes de que lo hiciese P3. En ese caso, P2 hace B=1 antes de que en la caché de P3 se escriba A=1. Entonces, P3 entra al condicional porque B=1, pero todavía no tiene escrito que A=1. Así que escribe reg1=0.
3) En el modelo SC hemos dicho que seguro que copia=1. En el caso en el que se relajan W->W podría ocurrir que P1 escribiese k=1 antes que A=1. Entonces, P2 saldría del bucle antes de que se escribiese A=1 y, por tanto, copia =0.
   En el caso en el que se relaja R->R podría ocurrir que P2 leyera copia=A antes de que se leyese si k==0. Es decir, no se asegura que P1 haya hecho ninguna de sus instrucciones. Por tanto, podría ocurrir que copia=0.

<u>Distinguir entre los diferentes modelos de consistencia relajados</u>

- Modelo que **relaja W->R**
  - Estos modelos permiten que en las operaciones ejecutadas en un procesador las **lecturas adelanten a escrituras**, pero evitando problemas de dependencias RAW.
  - Permite ocultar latencias de escritura.
  - Cuando sea necesario, se pueden utilizar para garantizar un resultado correcto, instrucciones 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 las lecturas posteriores.
    - La implementación del procesador permite que se pueda obtener el valor de una lectura directamente del buffer de escritura. Así, **un procesador puede leer antes que el resto de procesadores una escritura propia**.
  - Algunos sistemas difieren en cuanto a la atomicidad en los accesos de escritura (permiten que un procesador pueda ver el resultado de un acceso de escritura de otro procesador antes que el resto de procesadores).
    Por ello, otros sistemas fuerzan un acceso atómico, implementando accesos de **lectura-modificación-escritura atómicos**.

- Modelo que **relaja W->R y W->W**
  - Estos modelos, además, relajan el orden entre escrituras (a distintas direcciones).
    - Pueden permitir que el hardware 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 procesadores Sparc de Sun, el procesador puede leer por anticipado una escritura propia (del buffer de escritura), pero no puede leer por anticipado una escritura de otro. Así se garantizan los dos órdenes.

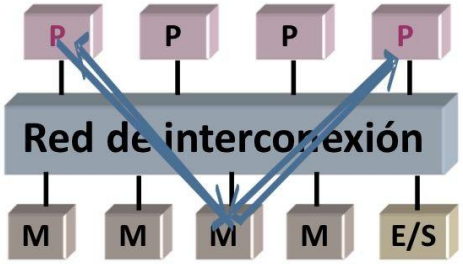
- Modelo de **reordenación débil**
    - Este modelo **relaja todos los órdenes**:
        - W->R
        - W->W
        - R->W
        - R->R
    - Se basa en mantener el orden entre accesos sólo en los puntos de sincronización del código. Para forzar el orden entre dos operaciones, el programador debe etiquetar alguna de ellas como operación de sincronización.
    - Si S es una operación de sincronización:
        - Las operaciones de acceso a memoria anteriores a la instrucción con etiqueta de sincronización, se deben completar antes que la operación de sincronización (WR->S)
        - Una operación etiqueta como de sincronización se debe completar antes que las operaciones de acceso a memoria posteriores (S->WR)
    - Un ejemplo es PowerPC

- Modelo de consistencia de liberación
    - Este modelo relaja todos los órdenes:
        - W->R
        - W->W
        - R->W
        - R->R
    - Tiene en cuenta que hay dos tipos de operaciones de sincronización: adquisición (leer variable compartida) y liberación (escribir variable compartida).
    - Mantiene los siguientes órdenes en los accesos:
        - Orden entre operación de adquisición y cualquier operación posterior (SA->WR)
        - Orden entre cualquier operación de acceso a memoria y una operación de liberación posterior (WR->SL)
    - Un ejemplo son los Sistemas Itanium

Lección 10: Sincronización

Por qué es necesaria la sincronización en multiprocesadores

La comunicación entre procesos en multiprocesadores se realiza a través de la **memoria compartida**.Para la buena comunicación, se necesita sincronizar el acceso a variables compartidas.



La comunicación puede ser entre dos procesadores o entre varios.

- **Comunicación uno-a-uno**:
  - Se debe garantizar que **el proceso** que lee la variable compartida (**recibe**), lo haga **después de que el proceso** que **envía** haya escrito en la variable el dato a enviar.
  - En un bucle, se debe garantizar que no se envía un nuevo dato hasta que no se haya recibido el anterior.
  - No puede haber dos procesos accediendo a la misma dirección compartida (**exclusión mutua**).
  - Una **sección crítica** es una secuencia de instrucciones con una o varias direcciones compartidas que se deben acceder con exclusión mutua.

Secuencial	Paralela	
... **A**=valor; ... copia=**A**; ...	**P1** ... A=valor; ...	**P2** ... copia=A; ...

...
mov **A**,rax
...
mov rbx,**A**
...
	P1
	...
	mov A,rax
	...
		P2
		...
		mov rbx,A
		...

En el ejemplo de arriba vemos que se quiere conseguir que, suponiendo que no ocurre nada con el valor A, copia=valor. Esto va a suceder cuando P1 realice la instrucción antes que P2.

Sin embargo, podría ocurrir que P2 hiciese la instrucción antes que P1 y, el resultado, no fuese el esperado.

Para que no ocurra, **debemos sincronizar** los procesadores. Por ejemplo, podríamos poner en el código de P2: while(A==valor){ copia=A; }

- **Comunicación colectiva**:
  - Se debe coordinar el acceso de múltiples procesos a una variable compartida, de forma que escriban uno detrás de otro o lean cuando tengan disponibles los resultados definitivos.
  - Por tanto, también se necesita un acceso en **exclusión mutua**.

Secuencial	**Paralela** (sum=0)

for (i=0 ; i<n ; i++) {
  sum = sum + a[i];
}
printf(sum);
	for (*i*=*ithread* ; *i*<n ; *i*=*i*+nthread) {
	  sump = *sump* + a[*i*];
	}
	sum = **sum** + *sump*; /* SC, sum compart. */
	if (*ithread*==0) printf(**sum**); Race condition

En el ejemplo anterior se pretende sumar todos los valores del vector a.
Para hacerlo en paralelo, se reparten las iteraciones entre el número de threads:

- Si hay 4 threads y 7 iteraciones: el thread 0 hace las iteraciones 1 y 5; el thread 1 hace las iteraciones 2 y 6; el thread 2, las iteraciones 3 y 7; y el thread 4, la cuarta iteración.

Cada thread tiene su variable propia sump que suma sus iteraciones y luego se pretende sumar todas las variables sump en sum.
Sin embargo, podría ocurrir que en el momento de sumar sum=sump, dos threads escribieran a la vez en la variable sum. Por ejemplo:

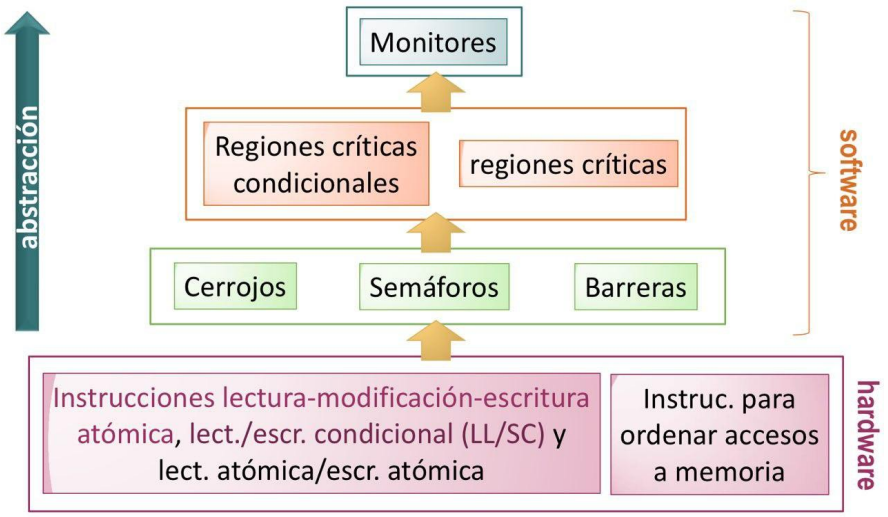
- Los threads 2 y 3 acceden a la vez a la dirección de m emoria de sum, cuando sum=1. Entonces, se hará sum=1+sump(2) y, a la vez, sum=1+sump(3). Cuando en realidad, lo que se pretendía que pasara es, sum=1+sump(2) y, posteriormente, sum=1+sump(2)+sump(3), para que se guardaran todas las sumas.

Para que esto no ocurra se necesita **sincronizar mediante cerrojos** (exclusión mutua).

- Por otro lado, podría ocurrir que el thread 0 llegase a la última instrucción antes de que la variable sum sumase todas las sump privadas. En ese caso, el thread 0 imprimiría una variable sum que no es la final.

Para que lo anterior no ocurra, se necesita **sincronizar mediante barreras**

<u>Describir primitivas para sincronización que ofrece el hardware</u>

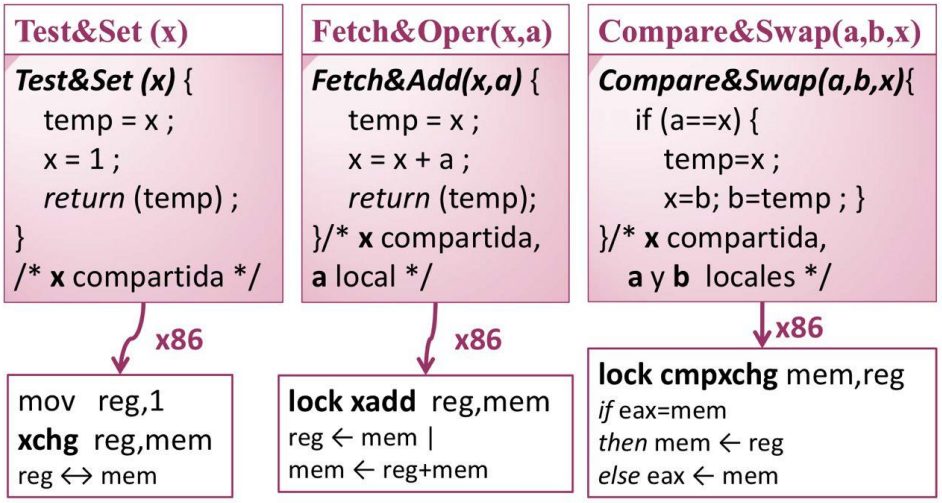


Para sincronizar los procesadores de un multiprocesador, el **hardware** ofrece:

1. Instrucciones de lectura-modificación-escritura atómica:
    a. **Test&Set(x)**: realiza de forma atómica sobre una variable compartida x, las siguientes operaciones:
        i. Lee en una variable local el contenido actual de x
        ii. Escribe un 1 en x
        iii. Devuelve como resultado el contenido de la variable local

    b. **Fetch&Oper(x,a)** -> *Oper se sustituye por Add, Increment…*: sobre una variable compartida x realiza las siguientes operaciones:
        i. Lee en una variable local el contenido actual de x
        ii. Escribe en x el resultado de realizar una operación (Oper) con el contenido de x y una variable local a
        iii. Devuelve el contenido de la variable local del punto i

    c. **Compare&Swap(a,b,x)**: realiza las siguientes operaciones:
        i. Lee el contenido actual de x
        ii. Si x==a, entonces se intercambian los contenidos de las variables b y x



2. Lectura y escritura condicional (LL/SC)
3. Lectura y escritura atómicas

El **software** ofrece:

1. Como mecanismos menos abstractos:
    a. **Cerrojos**: proporcionan una forma de asegurar exclusión mutua, es decir, que sólo un proceso puede acceder a una sección crítica. Para ello, dos funciones:
        i. Lock(k): un proceso adquiere el derecho a acceder a una sección crítica (cerrando el cerrojo k). Si otro proceso intenta adquirirlo, debe pasar una etapa de espera.
        ii. Unlock(k): Esta función libera a uno de los procesos que esperan el acceso a una sección crítica. Si no hay nadie esperando, se esperará hasta que otro proceso ejecute la operación lock(k).

Secuencial	Paralela

for (i=0 ; i<n ; i++) {
  sum = sum + a[i];
}
	for (*i=ithread* ; *i*<n ; *i*=*i*+nthread) {
	  sump = *sump* + a[*i*];
	}
	lock(k); //código de adquisición
	sum = **sum** + *sump*; /* SC, sum compart. */
	unlock(k); //código de liberación

El ejemplo está claro. Pero aun así, vamos a desarrollarlo un poco. Puede ocurrir que:

- Un proceso quiera entrar a la sección crítica con el cerrojo abierto (k=0). Entonces, cierra el cerrojo (k=1) y accede a la sección crítica. Cuando termine de ejecutarla, abre el cerrojo (k=0), que se queda abierto hasta que otro proceso lo cierre.
- Puede ocurrir que n procesos quieran entrar a la sección crítica cuando el cerrojo está cerrado (k=1), entonces deben esperar hasta que encuentren el cerrojo abierto (k=0). Cuando ocurra, sólo pasará a la sección crítica uno de los n procesos. Los n-1 procesos restantes seguirán esperando…
- Puede ocurrir que n procesos quieran entrar a la sección crítica cuando el cerrojo está abierto (k=0). Entonces, sólo un proceso pasa a la sección crítica, cierra el cerrojo (k=1) y los n-1 procesos se quedan esperando como en el apartado anterior.

Además, los cerrojos implementan dos mecanismos de espera: **espera ocupada** (el proceso consulta constantemente si se ha abierto el cerrojo) y **bloqueo** (el proceso queda suspendido para que el procesador atienda a otro proceso, cuando se ejecuta un unlock, se libera un proceso de los bloqueados).

    b. Semáforos
    c. **Barreras**: las barreras proporcionan un medio para asegurar que ningún proceso sobrepasa un punto de código hasta que todos los procesos hayan llegado a este punto.

2. En el siguiente nivel de abstracción:
    a. Regiones críticas condicionales
    b. Regiones críticas
3. El mecanismo más abstracto:
    a. Monitores

Implementar cerrojos simples

Como ya hemos mencionado en el apartado anterior, se implementan con una variable compartida k que toma dos valores: abierto (k=0) y cerrado (k=1).

La operación unlock escribe un 0 en la variable k. La operación lock lee la variable k y si k=0, escribe k=1.

Cabe destacar que **se debe garantizar el acceso en exclusión mutua a k y manteniendo un orden.**

lock (k)

```
lock(k) {
  while (leer-asignar_1-escribir(k) == 1) {} ;
} /* k compartida */
```

unlock (k)

```
unlock(k) {
  k = 0 ;
} /* k compartida */
```

En este ejemplo no se asegura que si el thread i está leyendo unlock, el thread j también lo lea antes de que el thread i escriba el k=1. No se está garantizando la exclusión mutua.

Para implementar correctamente los cerrojos simples se deben utilizar las funciones de lectura-escritura-modificación atómica vistas antes:

con Test&Set (x)

```
lock(k) {
  while (test&set(k)==1) {} ;
}
/* k compartida */
```

x86

```
lock:    mov  eax,1
repetir: xchg eax,k
         cmp  eax,1
         jz   repetir
```

En este caso, test&set(k) devuelve el valor del cerrojo cuando se lo encuentra, y va escribiendo k=1 continuamente.

con Fetch&Oper(x,a)

```
lock(k) {
  while (fetch&or(k,1)==1)  {};
}
/* k compartida */
```

true (1, cerrado)
false (0, abierto)

En este caso, fetch&or(k,1) devuelve el valor del cerrojo cuando se lo encuentra y escribe k=(kv1)=1, en todos los casos.

con Compare&Swap(a,b,x)

```
lock(k) {
  b=1
  do
    compare&swap(0,b,k) ;
  while (b==1);
}
/* k compartida, b local */
```

```
compare&swap(0,b,k){
    if (0==k) { b=k | k=b; }
}
```

En este caso, Compare&swap(0,b,k) lo que hace es comprobar si k==0, en este caso, se pone b=0 y se sale del bucle. En caso de que k==1, b=1 y se sigue en el bucle.

Implementar cerrojos con etiqueta

Los cerrojos con etiqueta fijan un orden FIFO en la adquisición del cerrojo.

```
lock (contadores)
contador_local_adq = contadores.adq;
contadores.adq = (contadores.adq + 1)  mod  max_flujos;
while (contador_local_adq != contadores.lib) {};

unlock (contadores)
contadores.lib = (contadores.lib + 1)  mod  max_flujos;
```

Funciona igual que el turno en la frutería.

- En la implementación de lock(k)

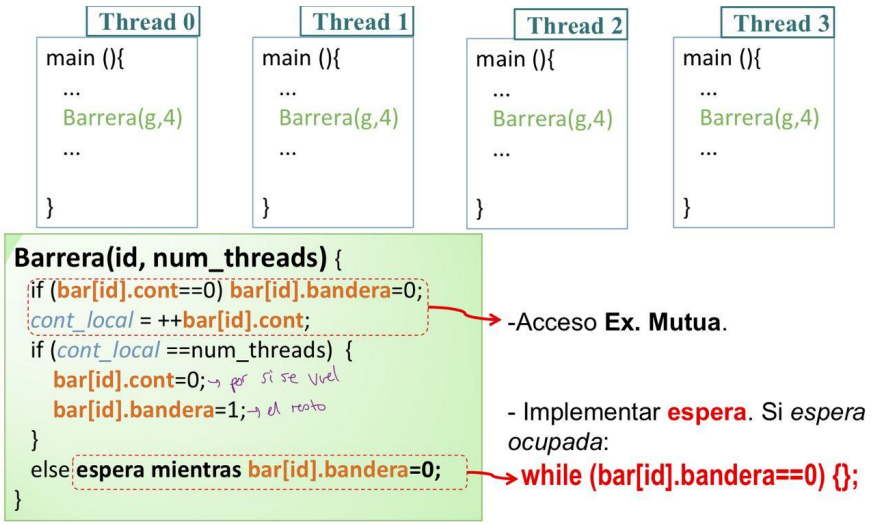
Cuando un flujo de instrucciones llega, coge turno (el turno se le guarda en la variable local *contador_local_adq*). Posteriormente, se incrementa *contadores.adq* para que el siguiente hilo que llegue coja el siguiente turno.
Va mirando la pantalla de los turnos (*contadores.lib*) hasta que le toque el suyo.

- En la implementación de unlock(k)

Cuando uno ya ha terminado de comprar la fruta (termina de ejecutar la sección crítica), en la pantalla se suma +1 para que pase el siguiente.

En este ejemplo, sigue sin garantizarse el acceso en exclusión mutua al contador de adquisición (dos threads pueden leer a la vez de memoria *contadores.adq* y coger el mismo turno). Además, habría que asegurar también el mismo orden para todos.
Para implementarlo correctamente, se podría utilizar Fetch&Increment.

Implementar barreras



Llega el primer thread a la barrera, el contador del flujo está a 0 (bar[id].cont==0), luego la bandera (variable compartida) también se pone a 0 (bar[id].bandera=0). En la siguiente línea, incrementa bar[id].cont y cont_local=1. Esto nos dice que ya hay un thread que ha

llegado a la barrera. Como cont_local != num_threads (4), se queda esperando hasta que la bandera sea 1.

Llega el siguiente thread a la barrera, el contador del flujo está a 0 (bar[id].cont==0), luego bar[id].bandera=0. Se incrementa su contador y cont_local=2. Se queda de nuevo en el bucle. Ya hay dos threads en el bucle.

Llega el siguiente thread y le ocurre lo mismo, ya hay tres threads en el bucle. Y cont_local=3.

Finalmente, llega el último thread. El contador del flujo está a cero, así que pone la bandera a 0 (bar[id].bandera=0). Se incrementa cont_local, que ahora está a 4. Pasa al if (cont_local==num_threads), por tanto, restablece su contador a 0 y restablece la bandera (variable compartida) a 1. Todos leen que la variable bandera ya no es 0, si no que es 1. Por tanto, el resto de threads ya salen del bucle y pueden seguir con el código.

Podría ocurrir que mientras uno de los threads espera, el SO decidiera suspenderlo para que haga otras instrucciones mientras. En ese caso, cuando se tenga que salir del bucle, al despertar, podría ocurrir que ya se haya puesto la bandera a 0 (porque hubiese otra barrera más adelante en el código a la que ya han llegado el resto de flujos). Entonces, este thread nunca saldría del bucle ya que la bandera nunca se pondría a 1 (porque la siguiente barrera nunca va a llegar a tener cont_local=nthreads). Por tanto, habría un **problema a la hora de la reutilización** de las banderas en varias barreras.

Barrera *sense-reversing*

```
Barrera(id, num_procesos) {
  bandera_local = !(bandera_local)   //se complementa bandera local
  lock(bar[id].cerrojo);
     cont_local = ++bar[id].cont       //cont_local es privada
  unlock(bar[id].cerrojo);
  if (cont_local == num_procesos)  {
     bar[id].cont = 0;                    //se hace 0 el cont. de la barrera
     bar[id].bandera = bandera_local; //para liberar thread en espera
  }
  else  while (bar[id].bandera != bandera_local) {};   //espera ocupada
}
```

Este ejemplo implementa barreras sin problema de reutilización. Para ello, usa cerrojos.

En este caso, cada vez que se reutiliza la bandera, los procesos para salir esperan una condición distinta. Se va a cambiar la condición para la liberación entre usos consecutivos de la barrera. Si en la última utilización han esperado para salir a que la bandera sea 1, en la siguiente vana esperar para salir que ésta sea 0, y en la siguiente nuevamente esperarán a que sea 1.