

2º curso / 2º cuatr.

Grados sen  
Ing. Informática

# Arquitectura de Computadores

## Tema 3

# Arquitecturas con paralelismo a nivel de thread (TLP)

Material elaborado por Mancia Anguita y Julio Ortega  
Profesores: Mancia Anguita, Maribel García y Christian Morillas



*ugr*

Universidad  
de Granada



# Bibliografía Tema 3

## ➤ Fundamental

- M. Anguita, J. Ortega, “Fundamentos y Problemas de Arquitectura de Computadores”, Avicam, 2016. (Cap. 3)
- J. Ortega, M. Anguita, A. Prieto, “Arquitectura de Computadores”, Thomson, 2005. (Cap.10)

# Lecciones

- Lección 7. Arquitecturas TLP
  - Clasificación y estructura de arquitecturas con TLP explícito y una instancia del SO
  - Multiprocesadores
  - Multicores
- Lección 8. Coherencia del sistema de memoria
- Lección 9. Consistencia del sistema de memoria
- Lección 10. Sincronización

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

- Multiprocesador
  - Ejecutan varios threads en paralelo en un **computador** con varios cores/procesadores.
    - Diversos niveles de empaquetamiento: dado, encapsulado, placa, chasis y sistema.
- Multicore o multiprocesador en un chip o CMP (*Chip MultiProcessor*)
  - Ejecutan varios threads en paralelo en un **chip de procesamiento** multicore (cada thread en un core distinto)
- Core multithread
  - **Core** que modifican su arquitectura ILP para ejecutar threads concurrentemente o en paralelo

# Multiprocesadores. Criterio de clasificación: nivel de empaquet./conexión

Sistema



SGI Altix 4700

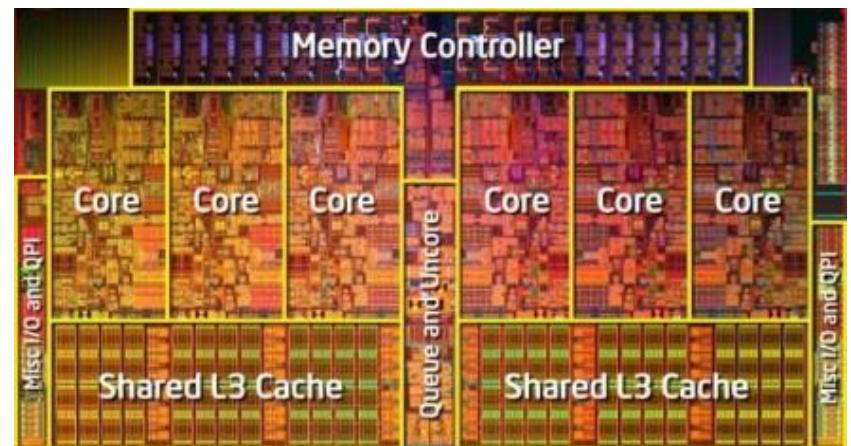
<http://www.sgi.com/products/remarketed/servers/altix4700.html>

Armario  
(*cabinet*)

Placa  
(*board*)  
chip



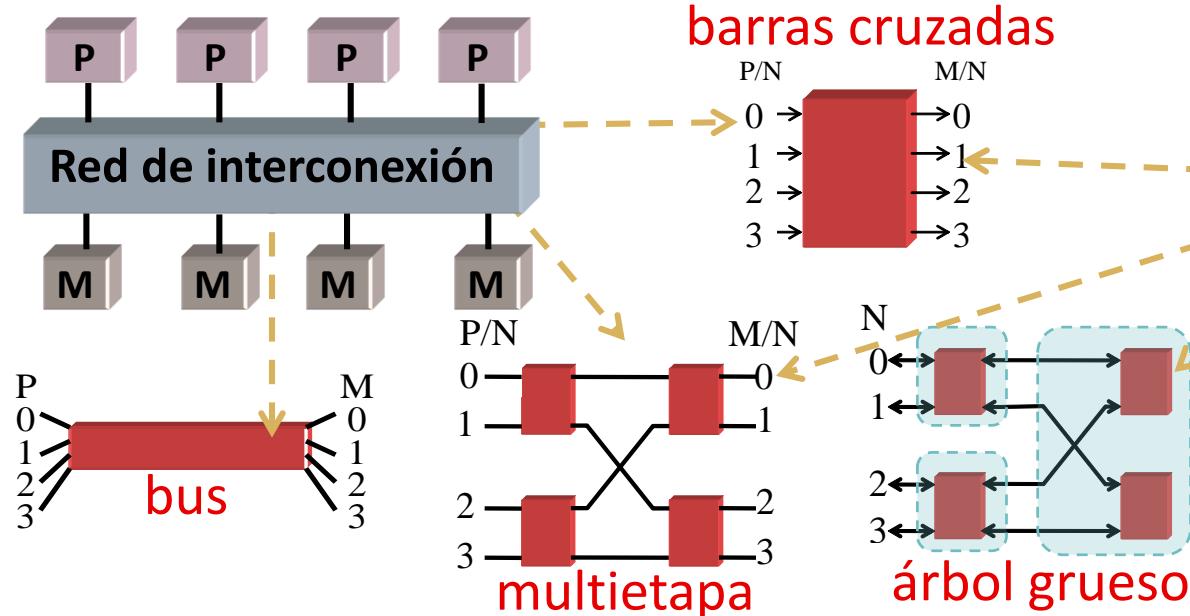
Multicore



# Multiprocesadores. Criterio clasificación: sistema de memoria (Lección 1)

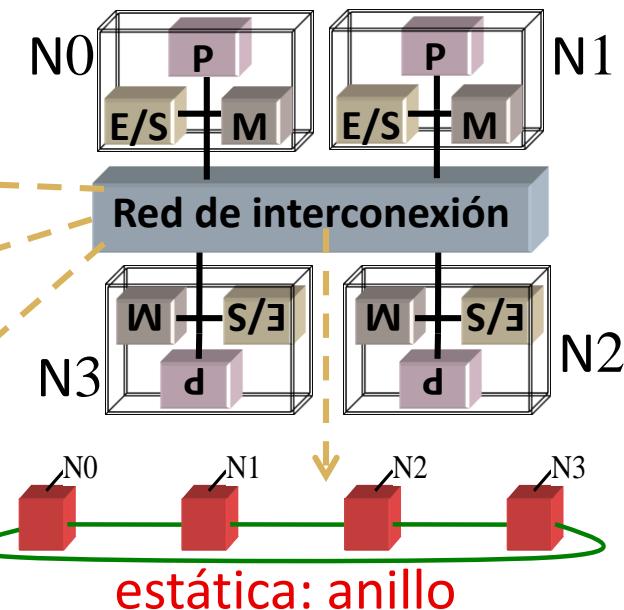
## Multiprocesador con memoria centralizada (UMA)

- Mayor latencia - Poco escalable

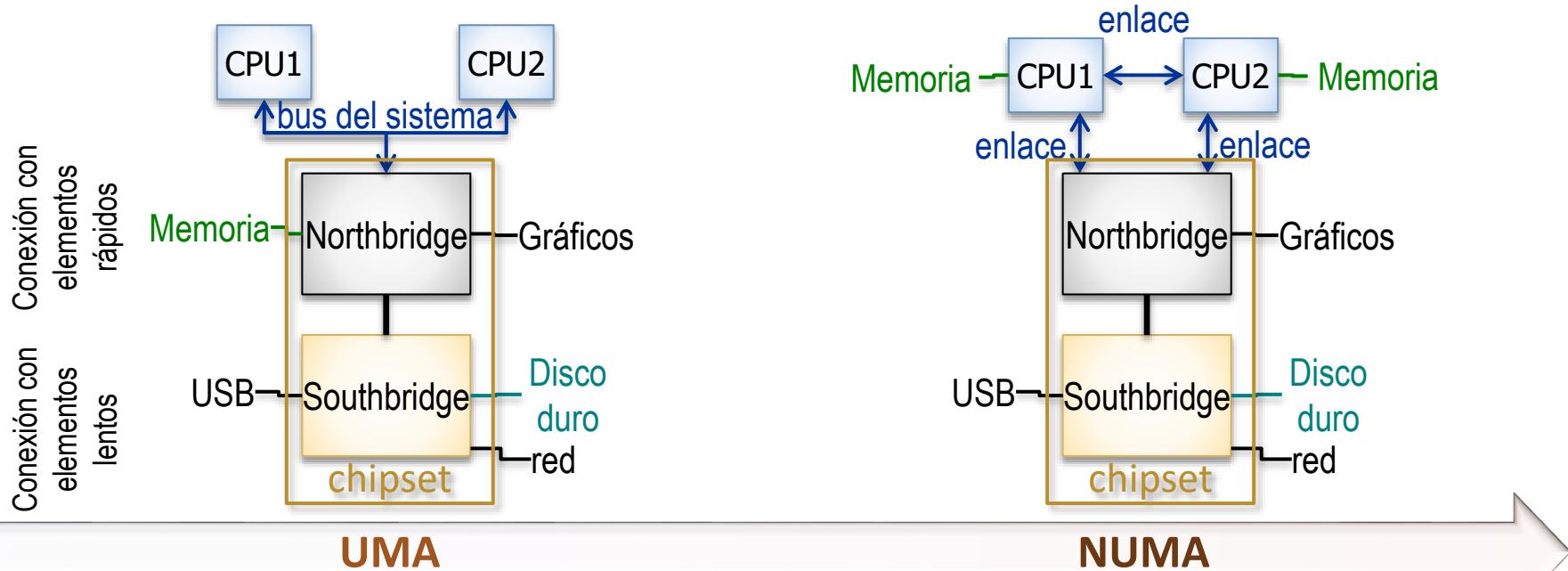


## Multiprocesador con memoria distribuida (NUMA)

- Menor latencia - escalable pero requiere para ello distribución de datos/código



# Multiprocesador en una placa: evolución de UMA a NUMA



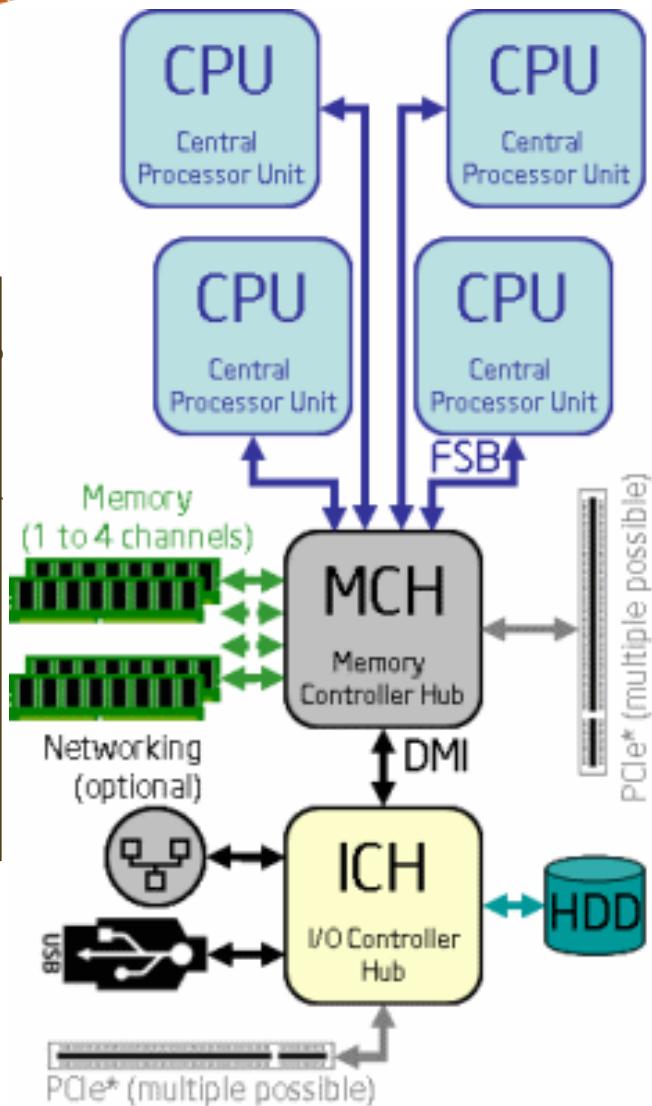
- ❖ Controlador de memoria en chipset (*Northbridge* chip)
- ❖ Red: bus (medio compartido)

- ❖ Controlador de memoria en chip del procesador
- ❖ Red: enlaces (conexiones punto a punto) y conmutadores (en el chip del procesador)
- ❖ Ejemplos en servidores:
  - AMD Opteron (2003): enlaces HyperTransport (2001)
  - Intel (Nehalem) Xeon 7500 (2010): enlaces QPI (*Quick Path Interconnect*, 2008)

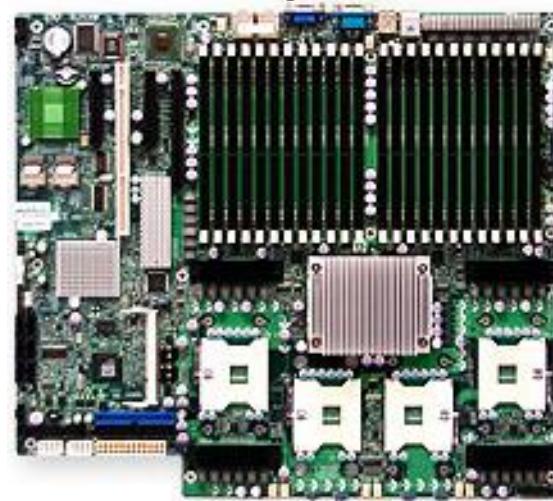
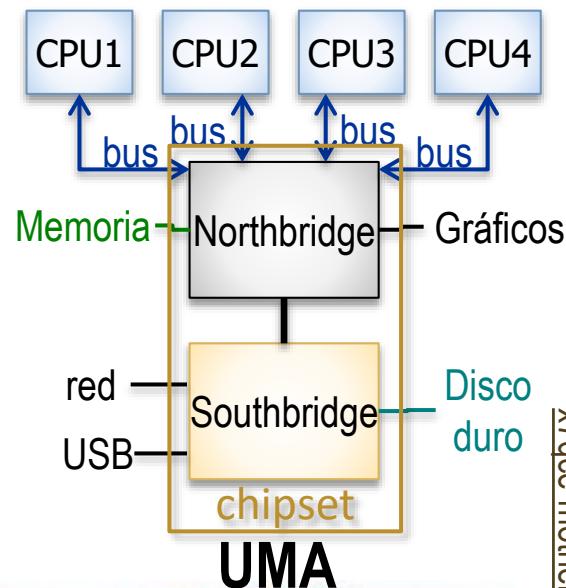
<http://www.intel.com/content/www/us/en/performance/performance-quickpath-architecture-demo.html>

# Multiprocesador en una placa: UMA con bus (Intel Xeon 7300)

[http://ark.intel.com/products/30792/Intel-Xeon-Processor-E7310-\(4M-Cache-1.60-GHz-1066-MHz-FSB\)#blockdiagrams](http://ark.intel.com/products/30792/Intel-Xeon-Processor-E7310-(4M-Cache-1.60-GHz-1066-MHz-FSB)#blockdiagrams)

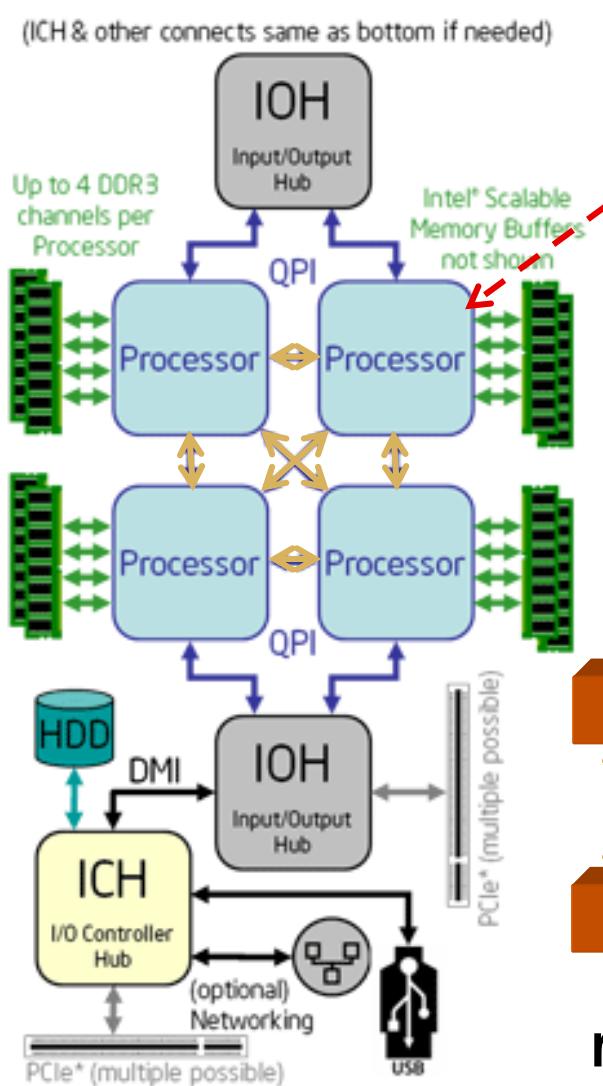


Conección con elementos rápidos  
Conección con elementos lentos

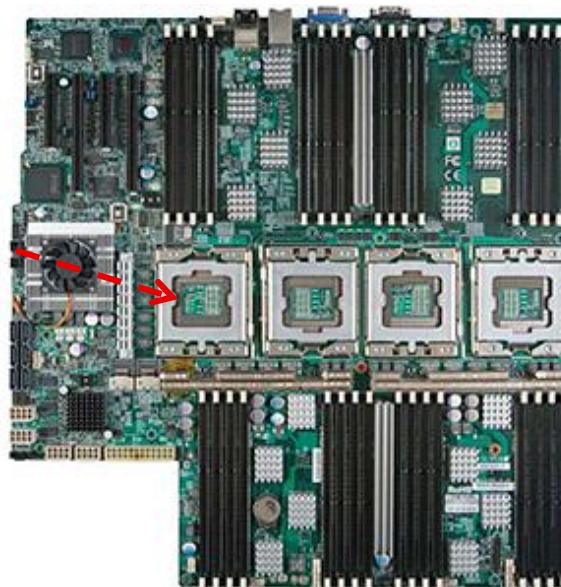
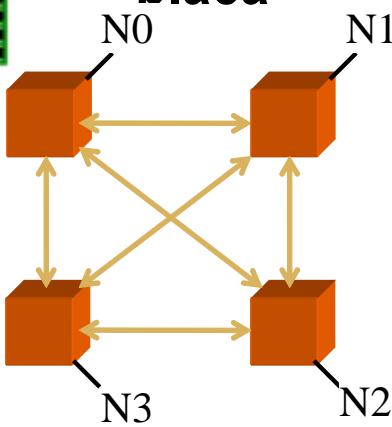


[http://shopper.cnet.com/motherboards/super-micro-x7qce-motherboard/4014-3049\\_9-32768194.html](http://shopper.cnet.com/motherboards/super-micro-x7qce-motherboard/4014-3049_9-32768194.html)

# Multiprocesador en una placa: CC-NUMA con red estática (Intel Xeon 7500)



# Diagrama de bloques de la placa



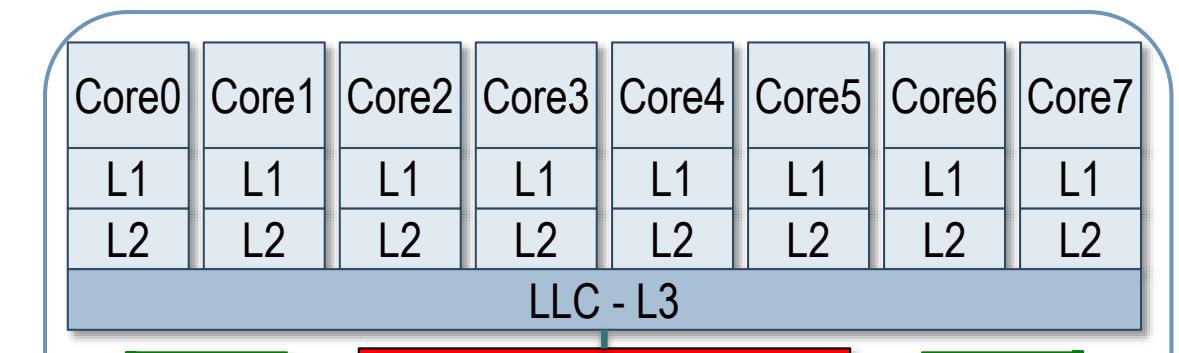
# Placa para Intel Xeon 7500

# Contenido Lección 7

- Clasificación y estructura de arquitecturas con TLP explícito y una instancia del SO
- Multiprocesadores
- Multicores
  - Ejecutan varios threads en paralelo en un **chip de procesamiento** multicore (cada thread en un core distinto)

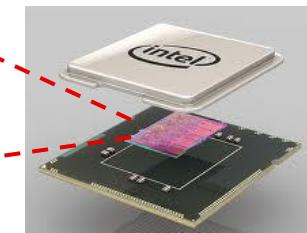
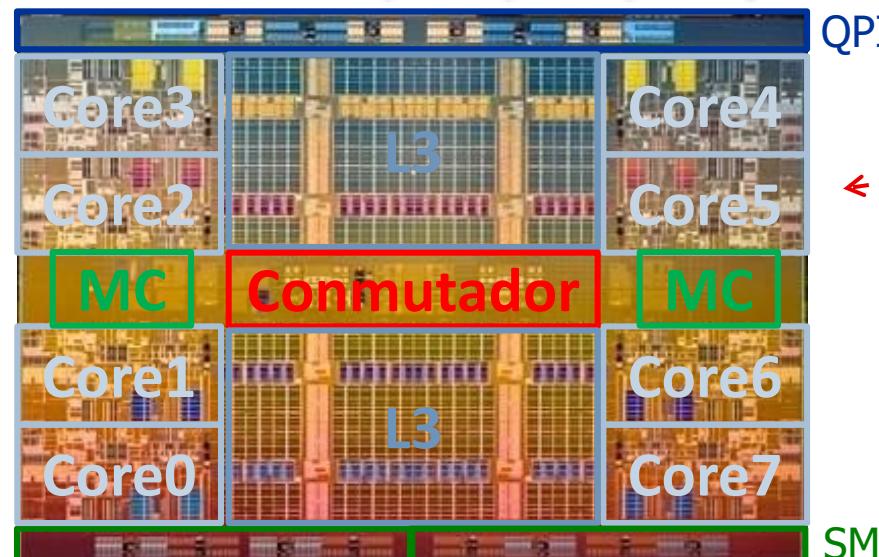
# Multiprocesador en un chip o Multicore o CMP (*Chip MultiProcessor*)

Intel Xeon 7500



32 KB Icache + 32KB Dcache  
8x256 KB  
12 a 24 MB

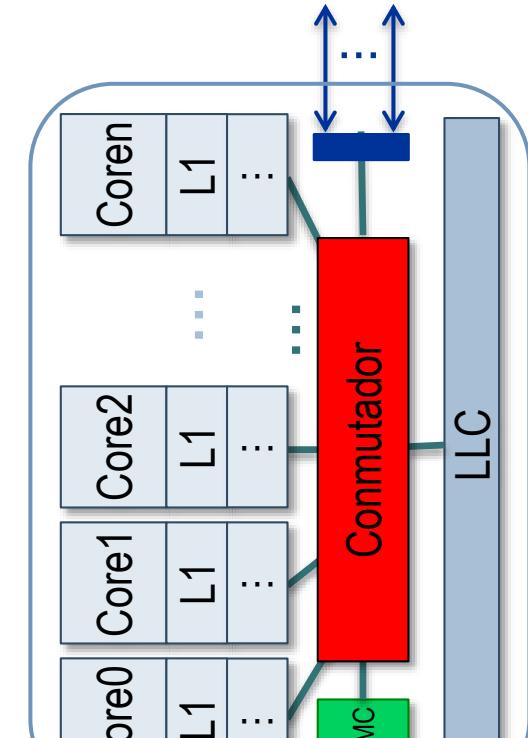
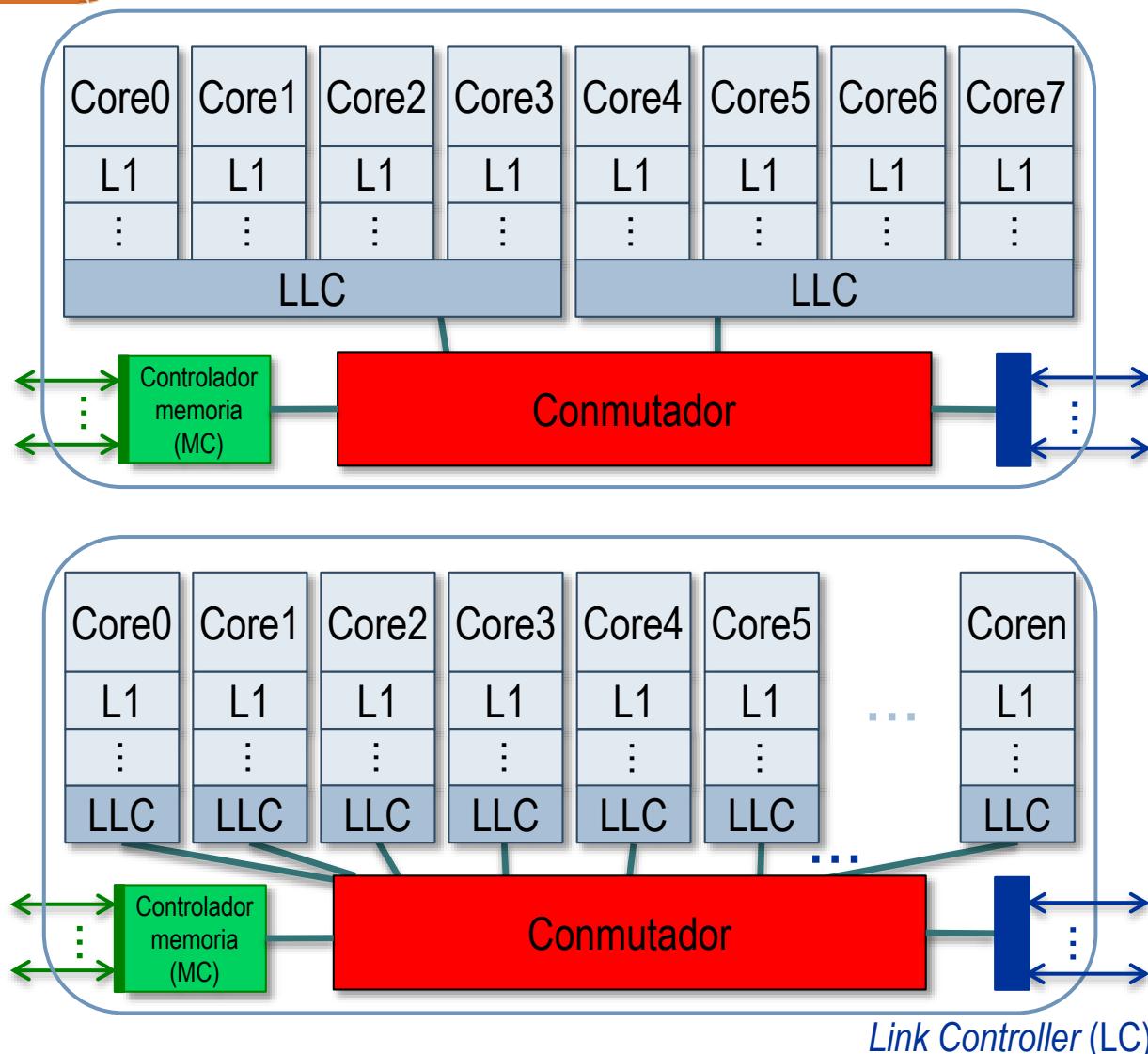
**Diagrama de bloques  
del chip**



**Chip o dado de silicio del chip de  
procesamiento Intel Xeon 7500**

[http://en.wikipedia.org/wiki/List\\_of\\_Intel\\_Xeon\\_microprocessors#22Beckton.22\\_2845\\_nm.29](http://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors#22Beckton.22_2845_nm.29)

# Multicore: otras posibles estructuras



# Para ampliar ...

## ➤ Webs

- An Introduction to the Intel® QuickPath Interconnect,  
<http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html>
- Intel® QuickPath Technology Animated Demo [119 K]  
<http://www.intel.com/content/www/us/en/performance/performance-quickpath-architecture-demo.html>

2º curso / 2º cuatr.

Grados sen  
Ing. Informática

# Arquitectura de Computadores

## Tema 3

# Arquitecturas con paralelismo a nivel de thread (TLP)

Material elaborado por Mancia Anguita y Julio Ortega  
Profesores: Mancia Anguita, Maribel García y Christian Morillas



*ugr*

Universidad  
de Granada



# Lecciones

- Lección 7. Arquitecturas TLP
- Lección 8. Coherencia del sistema de memoria
  - Sistema de memoria en multiprocesadores
  - Concepto de coherencia en el sistema de memoria: situaciones de incoherencia y requisitos para evitar problemas en estos casos
  - Protocolos de mantenimiento de coherencia: clasificación y diseño
  - Protocolo MSI de espionaje
  - Protocolo MESI de espionaje
  - Protocolo MSI basado en directorios con o sin difusión
- Lección 9. Consistencia del sistema de memoria
- Lección 10. Sincronización

# Objetivos Lección 8

- Comparar los métodos de actualización de memoria principal implementados en cache.
- Comparar las alternativas para propagar un escritura en protocolos de coherencia de cache.
- Explicar qué debe garantizar el sistema de memoria para evitar problemas por incoherencias.
- Describir 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.

# Bibliografía Lección 8

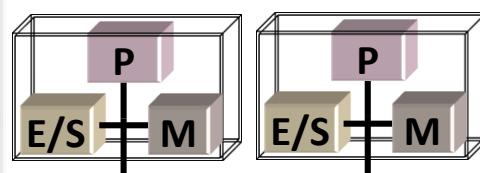
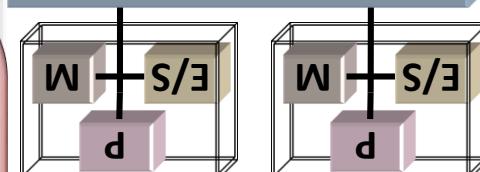
## ➤ Fundamental

- Cap.3. M. Anguita, J. Ortega. *Fundamentos y Problemas de Arquitectura de Computadores*. Ed. Avicam, 2016.
- Secc. 10.1. J. Ortega, M. Anguita, A. Prieto. *Arquitectura de Computadores*. Thomson, 2005. ESII/C.1 ORT arq

## ➤ Complementaria

- T. Rauber, G. Ründer. *Parallel Programming: for Multicore and Cluster Systems*. Springer 2010. Disponible en línea (biblioteca UGR): <http://dx.doi.org/10.1007/978-3-642-04818-0>

# Computadores que implementan en hardware mantenimiento de coherencia

<b>Multi-computadores</b> Memoria no compartida	<b>NORMA</b> <i>No Remote Memory Access</i>	nivel de sistema ( <i>Cluster</i> ), armario, chasis ( <i>blade server</i> )	Memoria físicamente distribuida   <b>Red de interconexión</b>	Escalabilidad +
	<b>NUMA</b> <i>Non-Uniform Memory Access</i>	NUMA (nivel de sistema, n. armario/chasis, n. placa)		
	<b>CC-NUMA</b> <i>Non-Uniform Memory Access</i>	CC-NUMA (nivel de armario: SGI Altix; nivel de placa)		
<b>Multi-procesadores</b> Memoria compartida Un único espacio de direcciones	<b>COMA</b>		Memoria físicamente centralizada   <b>Red de interconexión</b>	Escalabilidad -
	<b>UMA</b> <i>Uniform Memory Access</i>	<b>Coherencia por hardware</b> <i>SMP Symmetric MultiProcessor</i> (nivel de placa; nivel de chip: multicores como Intel Core i7, i5, i3)		

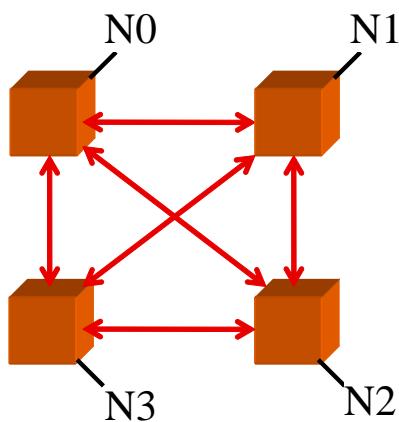
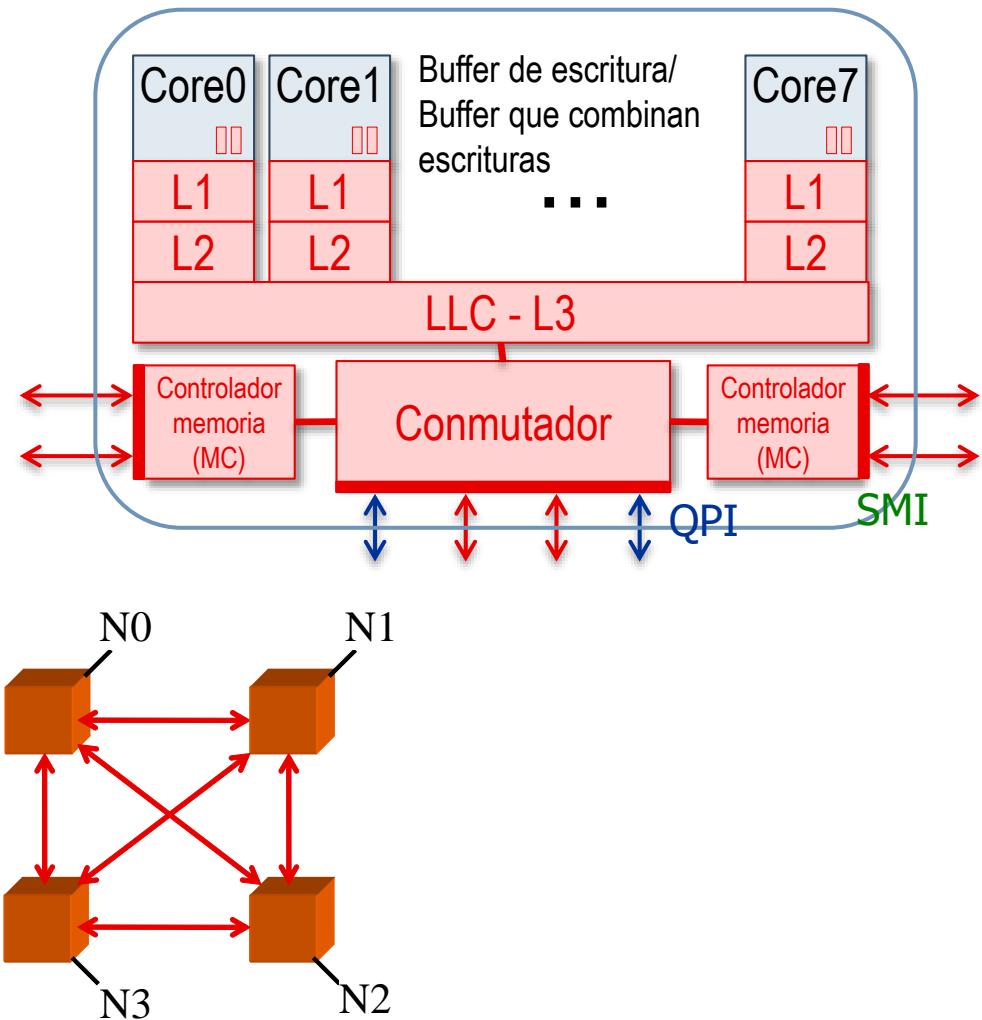
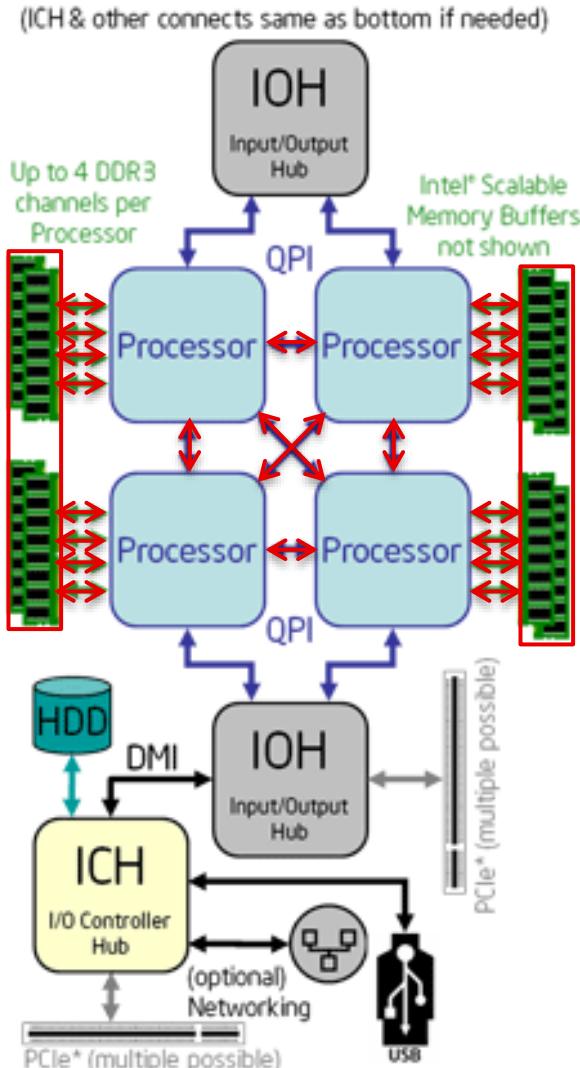
# Contenido Lección 8

- Sistema de memoria en multiprocesadores
- Concepto de coherencia en el sistema de memoria:  
situaciones de incoherencia y requisitos para evitar  
problemas en estos casos
- Protocolos de mantenimiento de coherencia:  
clasificación y diseño
- Protocolo MSI de espionaje
- Protocolo MESI de espionaje
- Protocolo MSI basado en directorios con o sin difusión

# Sistema de memoria en multiprocesadores

- ¿Qué incluye?
  - Caches de todos los nodos
  - Memoria principal
  - Controladores
  - Buffers:
    - Buffer de escritura/almacenamiento
    - Buffer que combinan escrituras/almacenamientos, etc.
  - Medio de comunicación de todos estos componentes (red de interconexión)
- La comunicación de datos entre procesadores la realiza el sistema de memoria
  - La lectura de una dirección debe devolver lo último que se ha escrito (desde el punto de vista de todos los componentes del sistema)

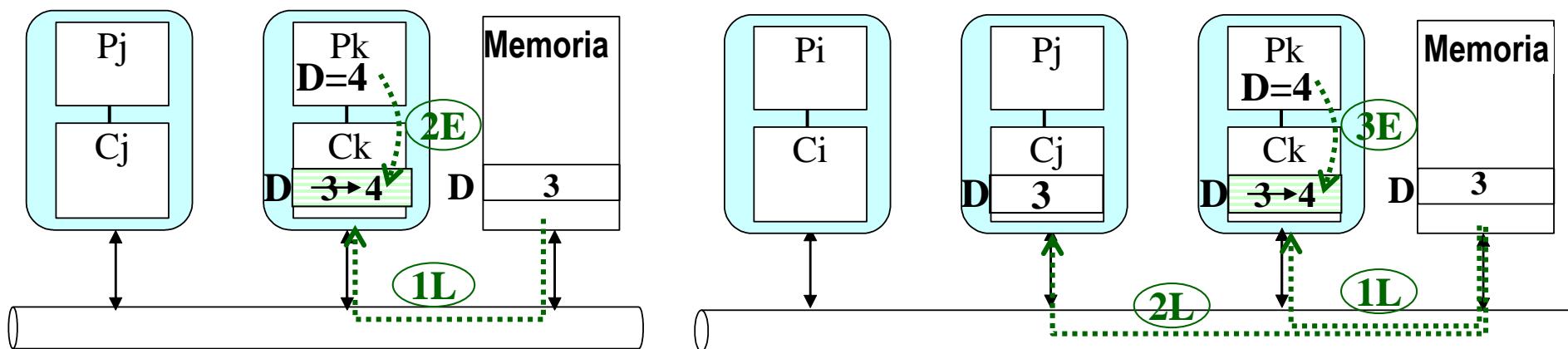
# Sistema de memoria



# Contenido Lección 8

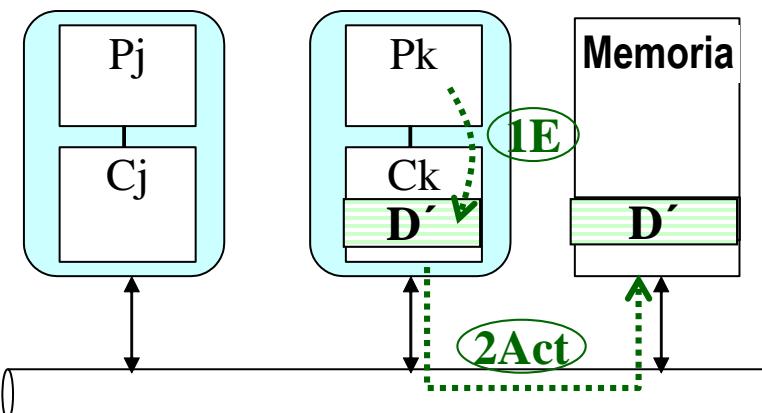
- Sistema de memoria en multiprocesadores
- Concepto de coherencia en el sistema de memoria:  
situaciones de incoherencia y requisitos para evitar  
problemas en estos casos
- Protocolos de mantenimiento de coherencia:  
clasificación y diseño
- Protocolo MSI de espionaje
- Protocolo MESI de espionaje
- Protocolo MSI basado en directorios con o sin difusión

# Incoherencia en el sistema de memoria



# Métodos de actualización de memoria principal implementados en caches

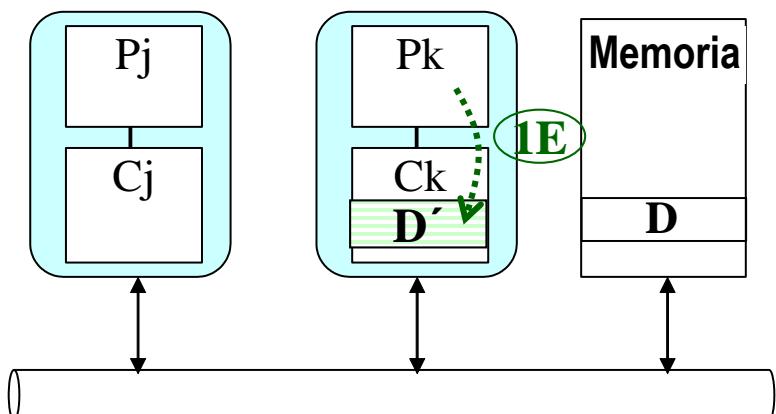
## 1. Escritura inmediata (*write-through*):



Cada vez que un procesador escribe en su cache escribe también en memoria principal

Por los principios de **localidad temporal** y **espacial** sería más rentable si se escribe todo el bloque una vez realizadas múltiples escrituras

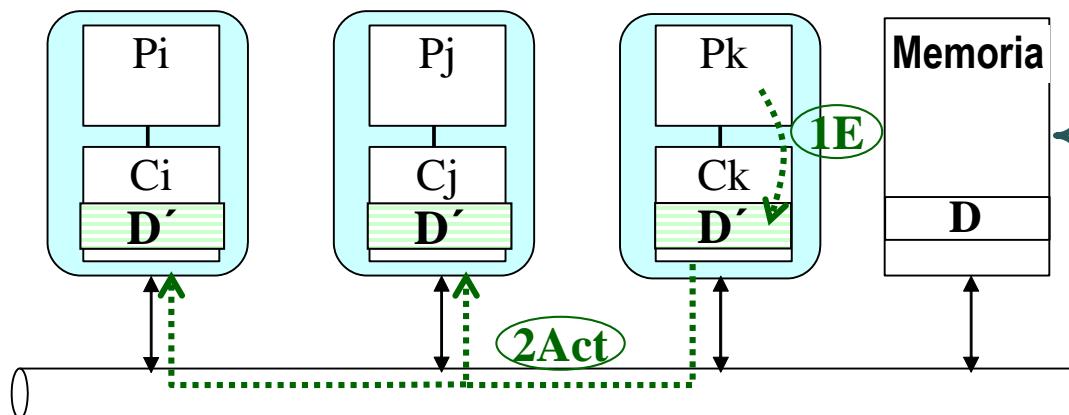
## 2. Posescritura (*write-back*):



Se actualiza memoria principal escribiendo todo el bloque cuando se **desaloja** de la cache

# Alternativas para propagar una escritura en protocolos de coherencia de cache

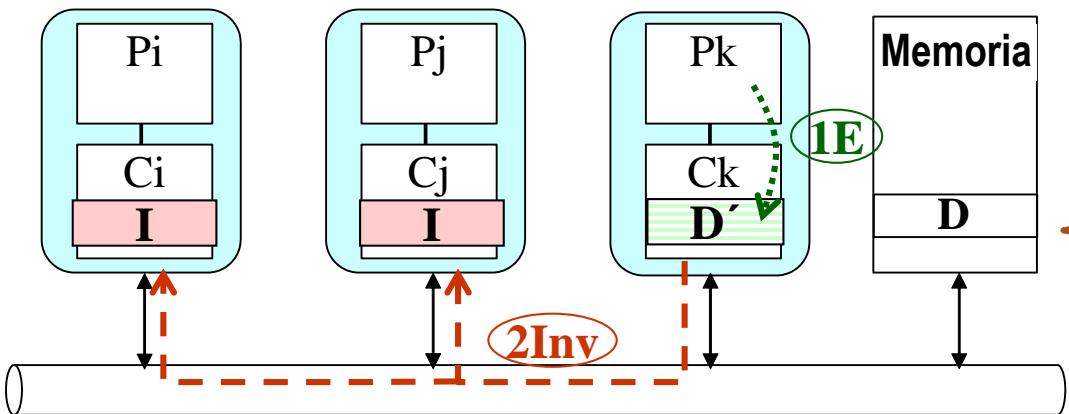
## 1. Escritura con actualización (*write-update*):



Cada vez que un procesador escribe en una dirección en su cache **se escribe** en las copias de esa dirección en otras caches

Para **reducir tráfico**, sobre todo si los datos están compartidos por pocos procesadores

## 2. Escritura con invalidación (*write-invalidate*):



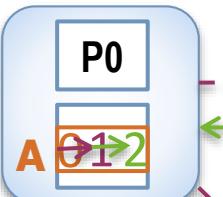
Antes que un procesador modifique una dirección en su cache **se invalidan** las copias del bloque de la dirección en otras caches

# Situación de incoherencia aunque se propagan las escrituras (usa difusión)

AC ATC

Orden para P0

- 1) A=1
- 2) **A=2**

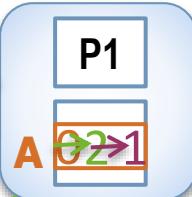


A=1

A=2

Orden para P1

- 1) A=2
- 2) **A=1**



A=2

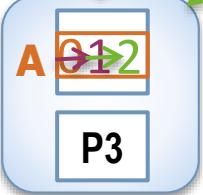
A=2

Orden generación:

- P0 1) A=1  
P1 2) A=2

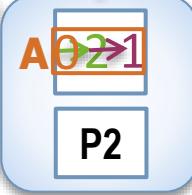
Orden para P3

- 1) A=1
- 2) **A=2**

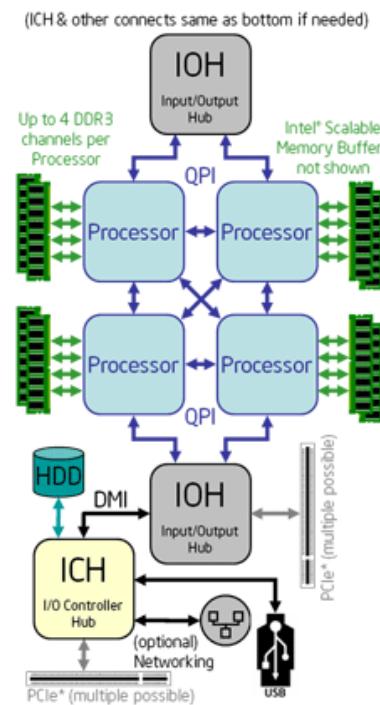


Orden para P2

- 1) A=2
- 2) **A=1**



A=2



- Contenido inicial de las copias de la dirección A en **calabaza**. P0 escribe en A un 1 y, después, P1 escribe en A un 2.
- Se utiliza **actualización** para propagar las escrituras (las propagación se nota con flechas)
- Llegan en distingo orden las escrituras debido al distinto tiempo de propagación (se está suponiendo que los Proc. están ubicados en la placa tal y como aparecen en el dibujo). En **cursiva** se puede ver el contenido de las copias de la dirección A tras las dos escrituras.
  - Se da una situación de incoherencia aunque se propagan las escrituras: P0 y P3 acaban con 2 en A y P1 y P2 con 1.

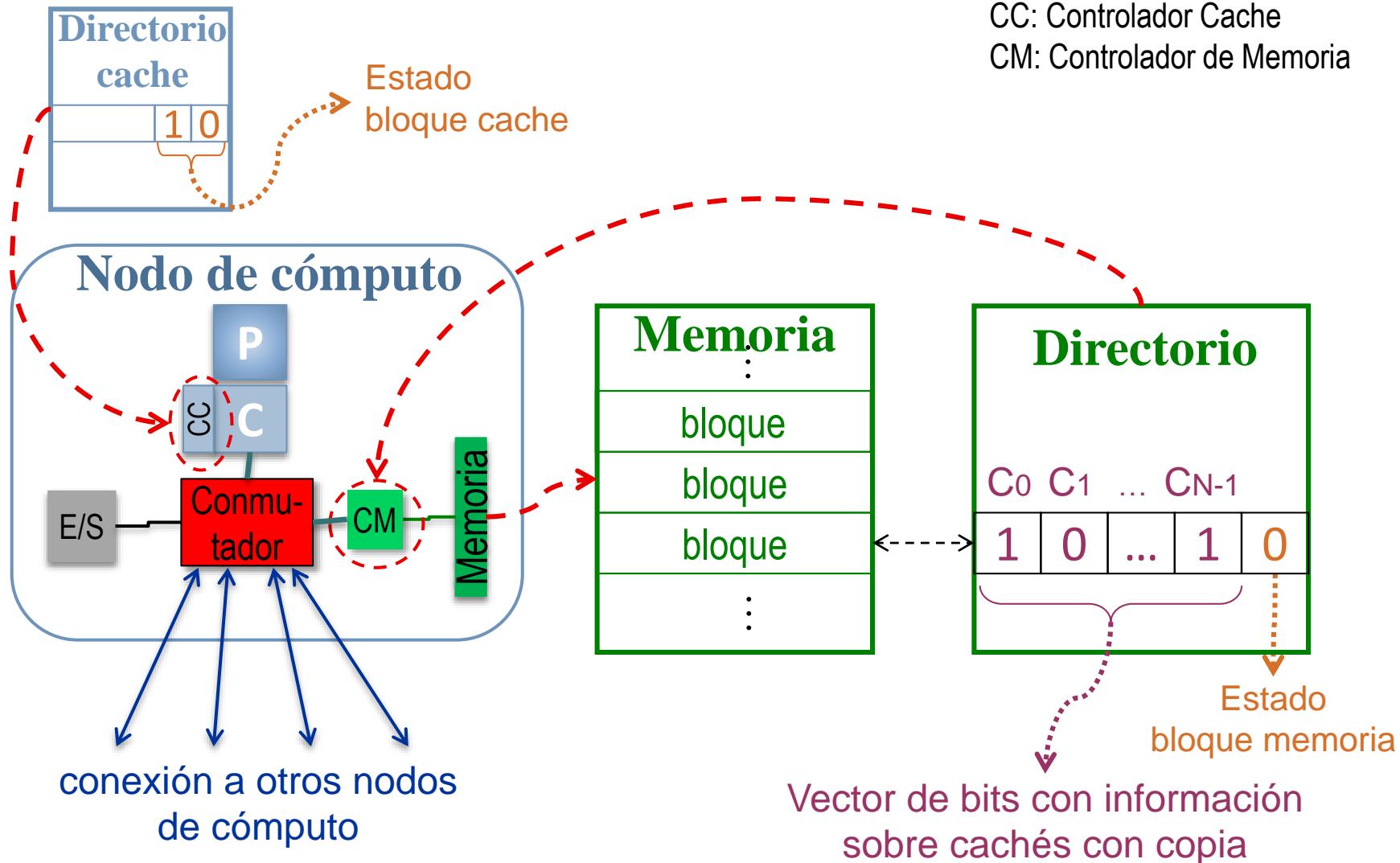
# Requisitos del sistema de memoria para evitar problemas por incoherencia I

- Propagar las escrituras en una dirección
  - La escritura en una dirección debe hacerse visible en un tiempo finito a otros procesadores
  - Componentes conectados con un bus:
    - Los paquetes de actualización/invalidación son visibles a todos los nodos conectados al bus (controladores de cache)
- Serializar las escrituras en una dirección
  - Las escrituras en una dirección deben verse 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)
  - Componentes conectados con un bus:
    - El orden en que los paquetes aparecen en el bus determina el orden en que se ven por todos los nodos.

# Requisitos del SM para evitar problemas por incoherencia II: la red no es un bus

- Propagar escrituras en una dirección
  - Usando difusión:
    - Los paquetes de actualización/invalidación se envían a todas las caches
  - Para conseguir mayor escalabilidad:
    - Se debería enviar paquetes de actualización/invalidación sólo a caches (nodos) con copia del bloque
    - Mantener en un directorio, para cada bloque, los nodos con copia del mismo
- Serializar escrituras en una dirección
  - El orden en el que las peticiones de escritura llegan a su *home* (nodo que tiene en MP la dirección) o al directorio centralizado sirve para serializar en sistemas de comunicación que garantizan el orden en las trasferencias entre dos puntos

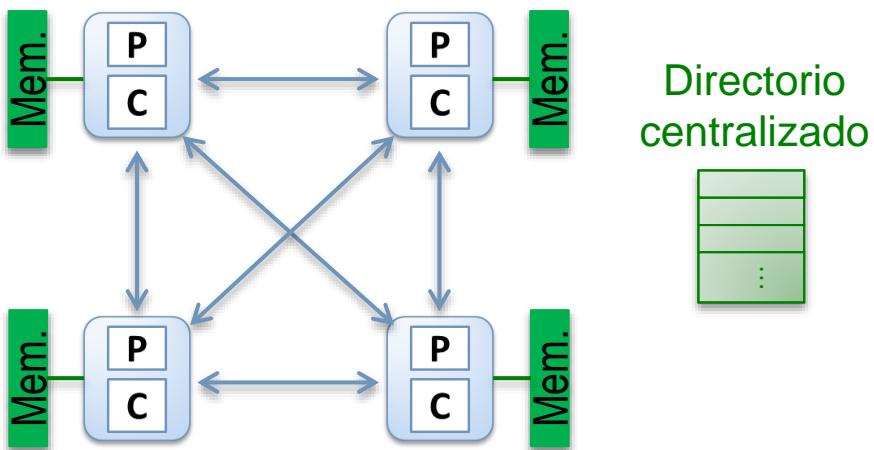
# Directorio de memoria principal



# Alternativas para implementar el directorio

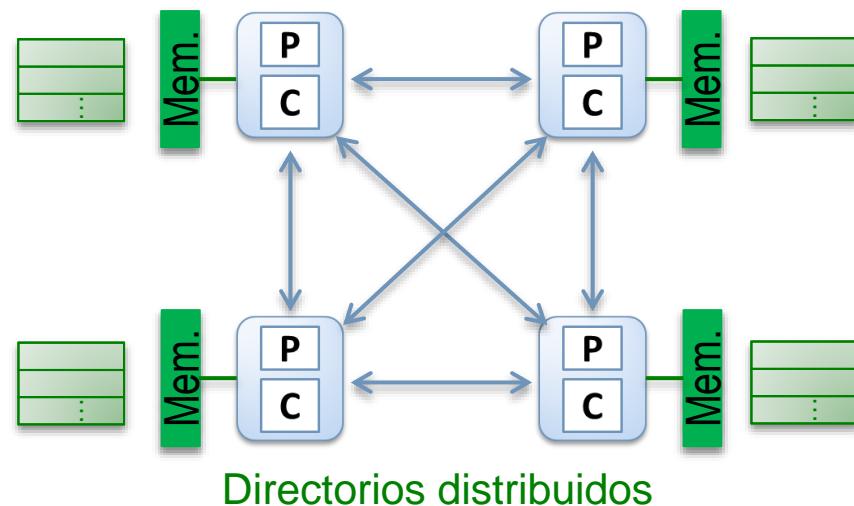
## Centralizado

- Compartido por todos los nodos
- Contiene información de los bloques de todos los módulos de memoria

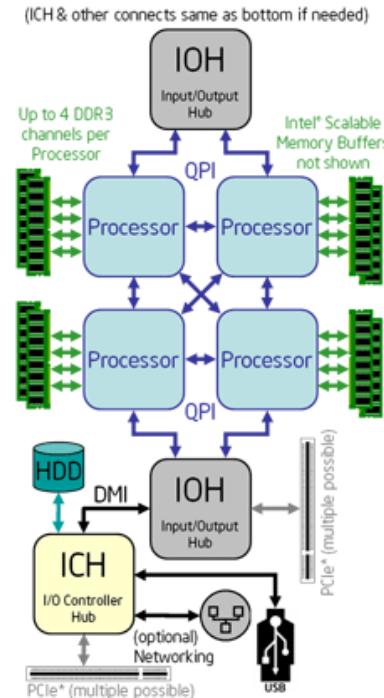
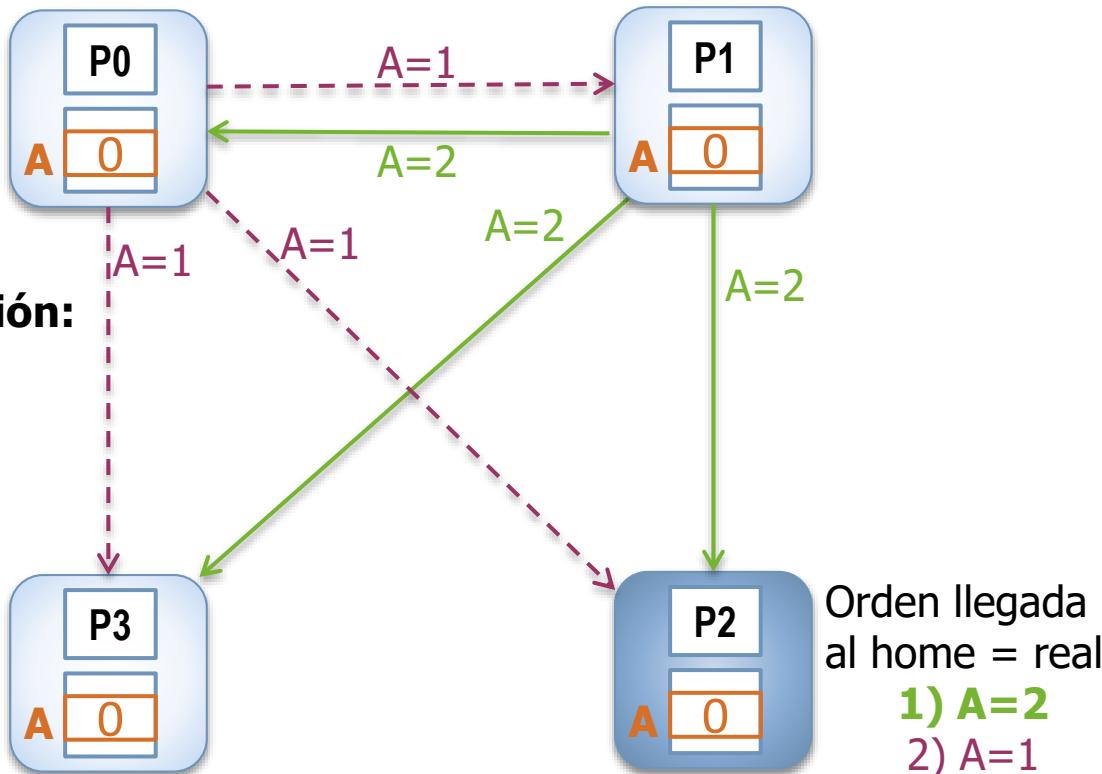


## Distribuido

- Las filas se distribuyen entre los nodos
- Típicamente el directorio de un nodo contiene información de los bloques de sus módulos de memoria

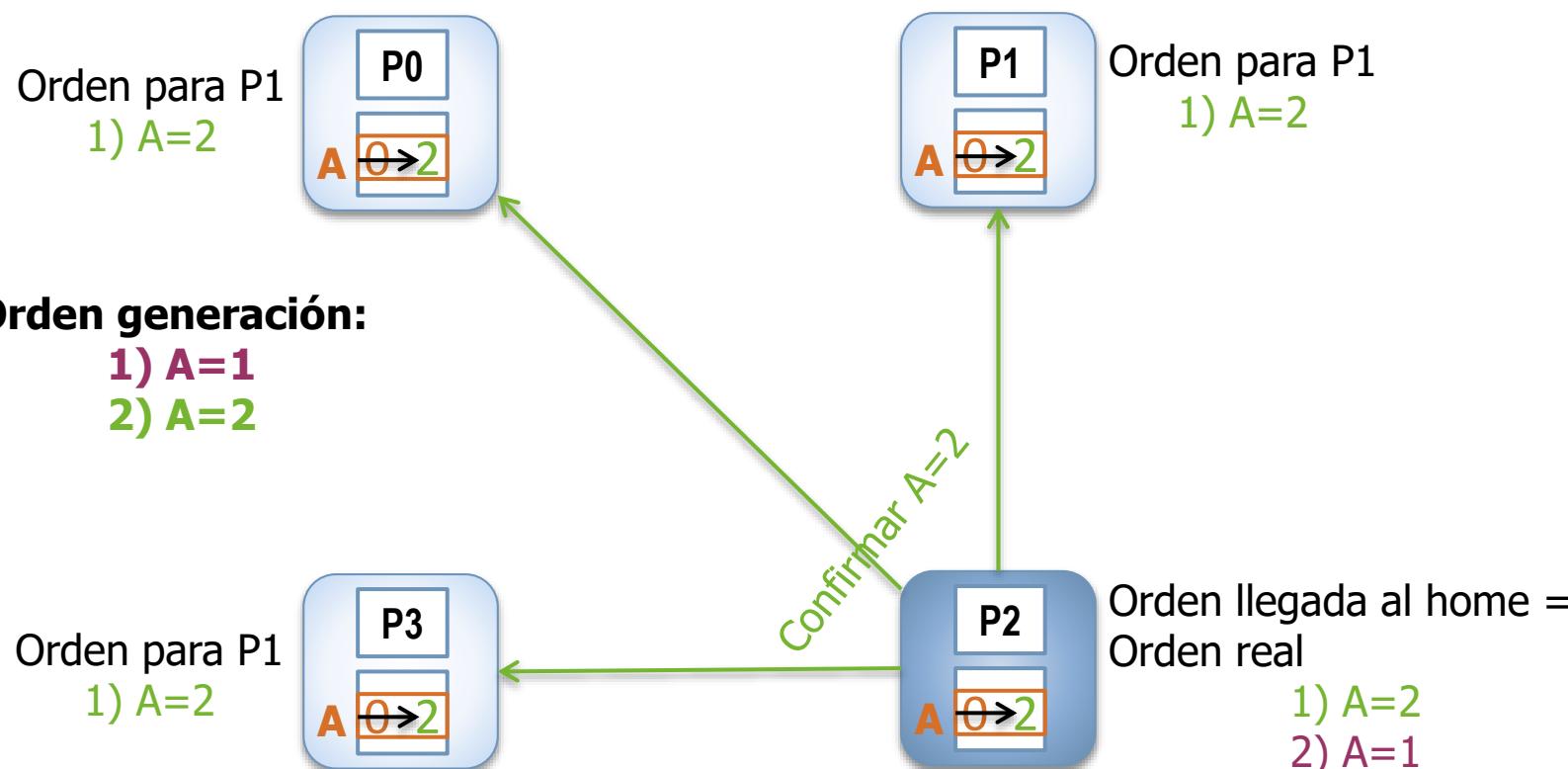


# Serialización de las escrituras por el home. Usando difusión I



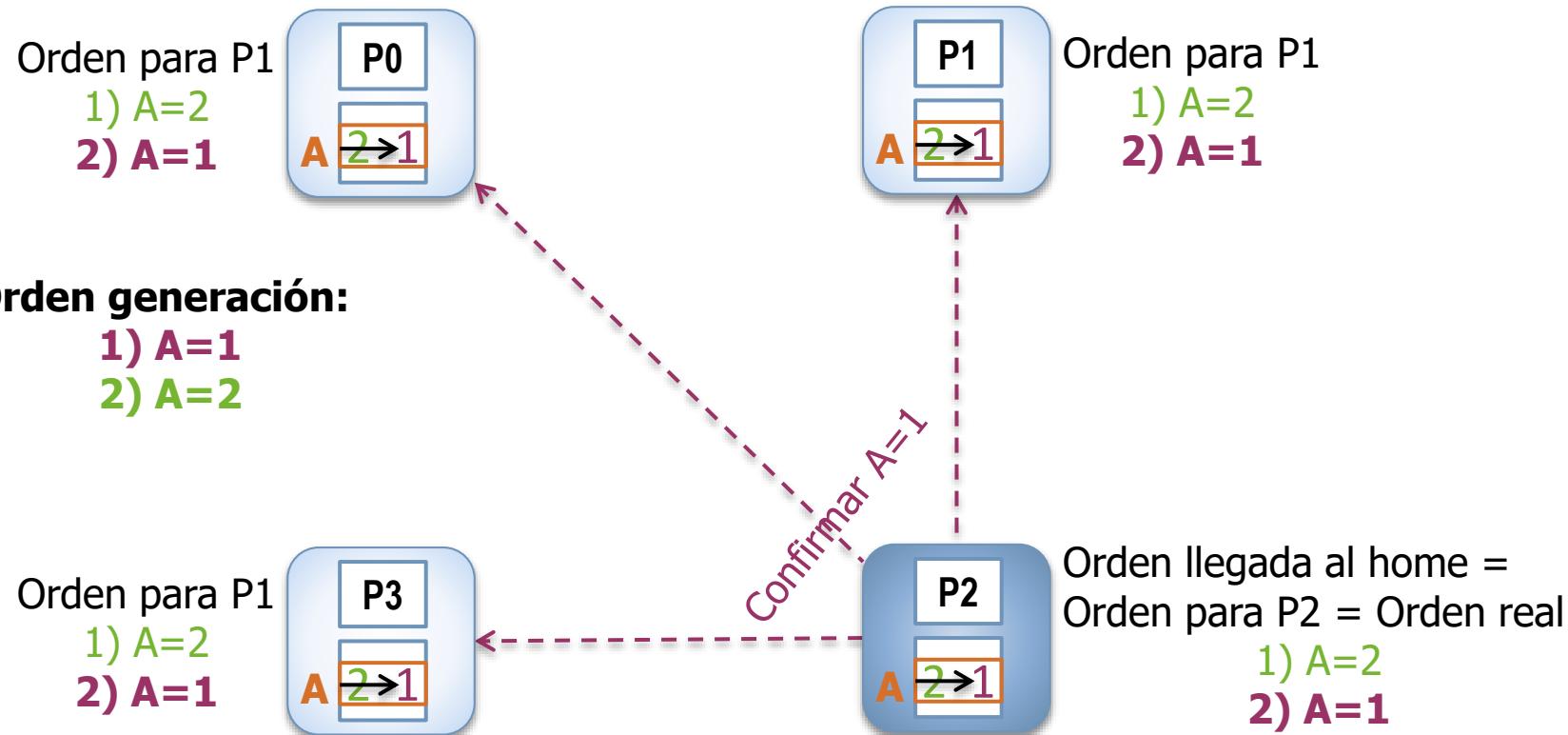
- Contenido inicial de las copias de la dirección A en **calabaza**. P0 escribe en A un 1 y, después, P1 escribe en A un 2.
- Se utiliza **actualización** para propagar las escrituras (las propagación se nota con flechas)
- El orden de llegada al home es el orden real para todos

# Serialización de las escrituras por el home. Usando difusión II



- Contenido inicial de las copias de la dirección A en **calabaza**
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas)

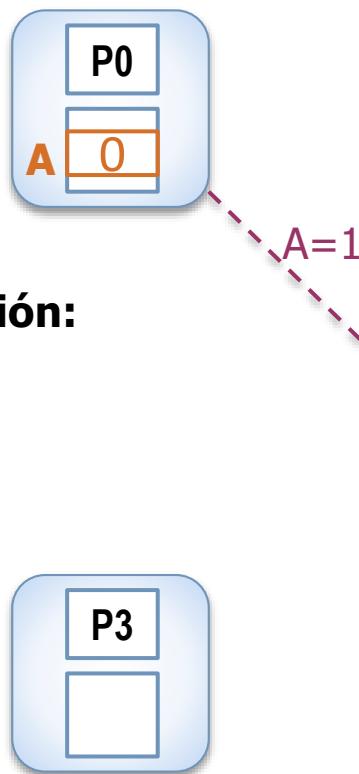
# Serialización de las escrituras por el home. Usando difusión III



- Se utiliza actualización para propagar las escrituras (la propagación se nota con flechas)

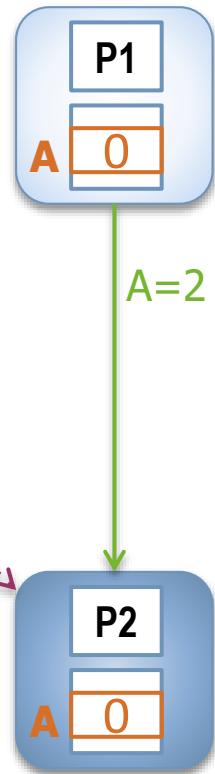
# Serialización de las escrituras por el home.

## Sin difusión y con directorio distribuido I



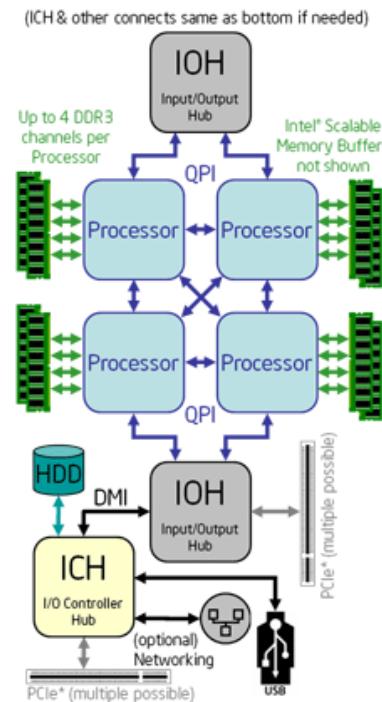
**Orden generación:**

- 1) A=1
- 2) A=2



Orden llegada al home = real

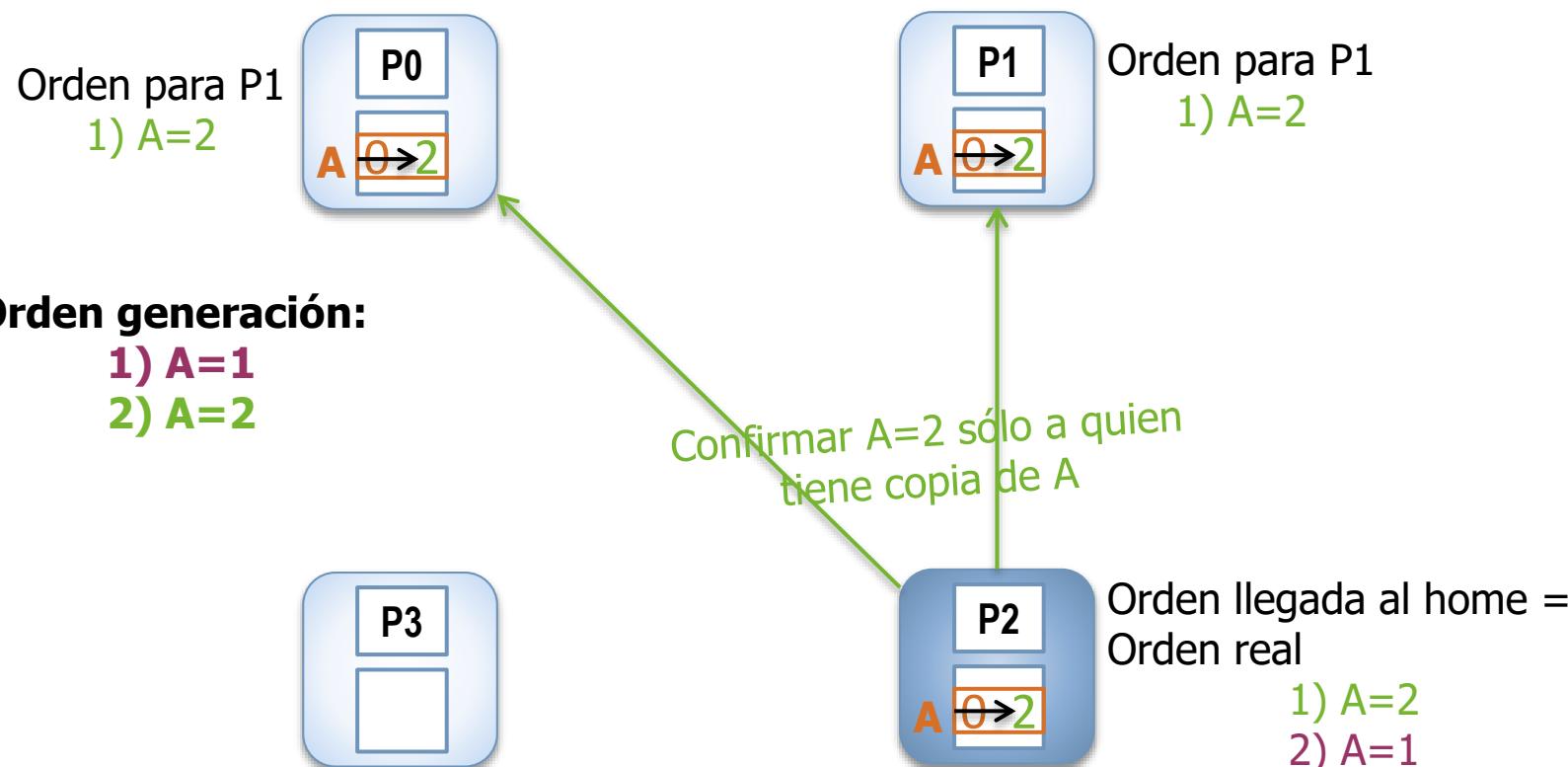
- 1) A=2
- 2) A=1



- Contenido inicial de las copias de la dirección A en **calabaza**. P0 escribe en A un 1 y, después, P1 escribe en A un 2.
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas)
- El orden de llegada al home es el orden real para todos

# Serialización de las escrituras por el home.

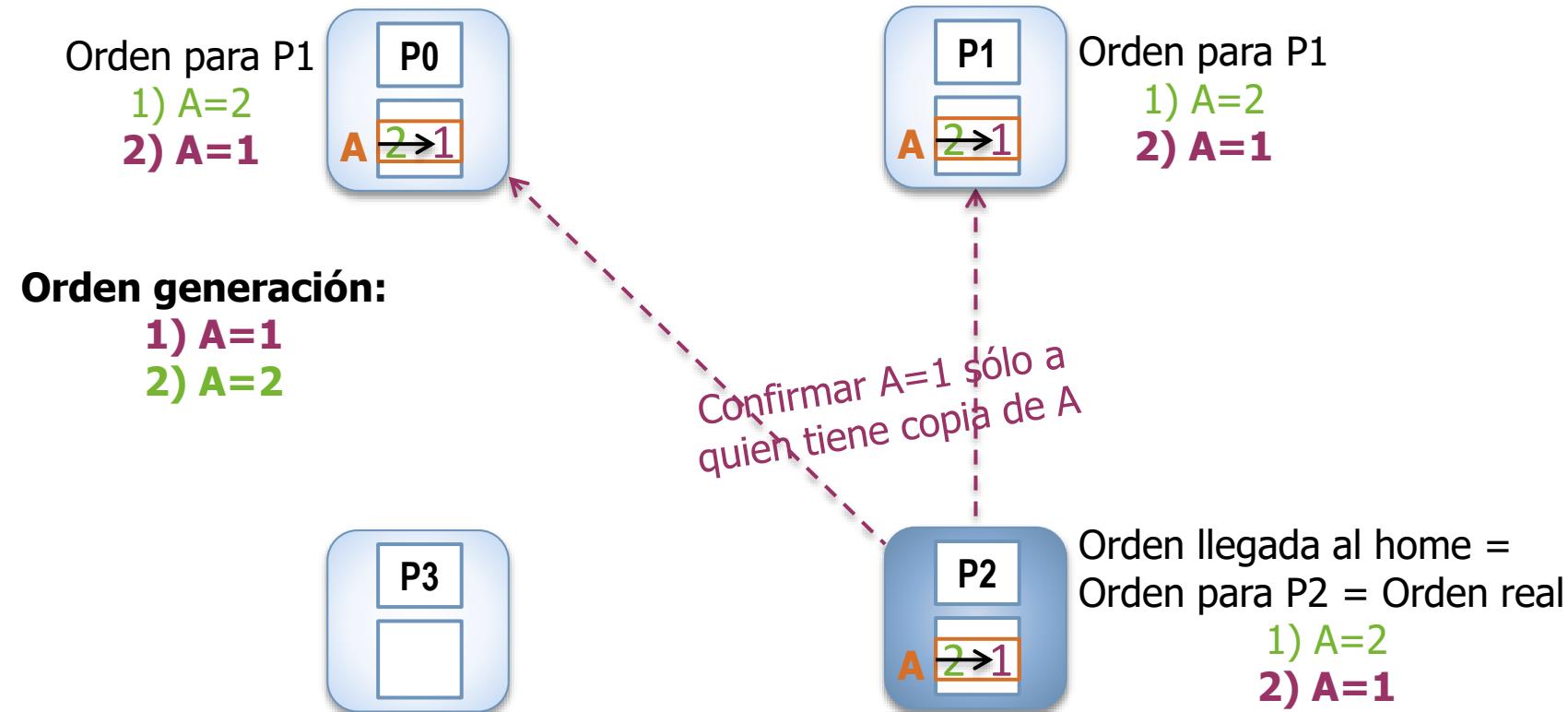
## Sin difusión y con directorio distribuido II



- Contenido inicial de las copias de la dirección A en **calabaza**
- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas)

# Serialización de las escrituras por el home.

## Sin difusión y con directorio distribuido III



- Se utiliza actualización para propagar las escrituras (las propagación se nota con flechas)

# Contenido Lección 8

- Sistema de memoria en multiprocesadores
- Concepto de coherencia en el sistema de memoria:  
situaciones de incoherencia y requisitos para evitar  
problemas en estos casos
- Protocolos de mantenimiento de coherencia:  
clasificación y diseño
- Protocolo MSI de espionaje
- Protocolo MESI de espionaje
- Protocolo MSI basado en directorios con o sin difusión

# Clasificación de protocolos para mantener coherencia en el sistema de memoria

- Protocolos de espionaje (snoopy)
  - Para buses, y en general sistemas con una difusión eficiente (bien porque el número de nodos es pequeño o porque la red implementa difusión).
- Protocolos basados en directorios.
  - Para redes sin difusión o escalables (multietapa y estáticas).
- Esquemas jerárquicos.
  - Para redes jerárquicas: jerarquía de buses, jerarquía de redes escalables, redes escalables-buses.

# Facetas de diseño lógico en protocolos para coherencia

- Política de actualización de MP:
  - escritura inmediata, posescritura, mixta
- Política de coherencia entre caches:
  - escritura con invalidación, escritura con actualización, mixta
- Describir comportamiento:
  - Definir posibles estados de los bloques en cache, y en memoria
  - Definir transferencias (indicando nodos que intervienen y orden entre ellas) a generar ante eventos:
    - lecturas/escrituras del procesador del nodo
    - como consecuencia de la llegada de paquetes de otros nodos.
  - Definir transiciones de estados para un bloque en cache, y en memoria

# Contenido Lección 8

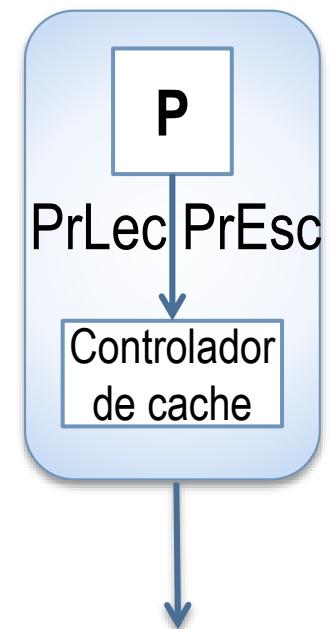
- Sistema de memoria en multiprocesadores
- Concepto de coherencia en el sistema de memoria:  
situaciones de incoherencia y requisitos para evitar  
problemas en estos casos
- Protocolos de mantenimiento de coherencia:  
clasificación y diseño
- Protocolo MSI de espionaje
- Protocolo MESI de espionaje
- Protocolo MSI basado en directorios con o sin difusión

# Protocolo de espionaje de tres estados (MSI) – posescritura e invalidación

- Estados de un bloque en cache:
  - **Modificado (M)**: es la única copia del bloque válida en todo el sistema
  - **Compartido (C,S)**: está válido, también válido en memoria y puede que haya copia válida en otras caches
  - **Inválido (I)**: se ha invalidado o no está físicamente
- Estados de un bloque en memoria (en realidad se evita almacenar esta información):
  - **Válido**: puede haber copia válida en una o varias caches
  - **Inválido**: habrá copia valida en una cache

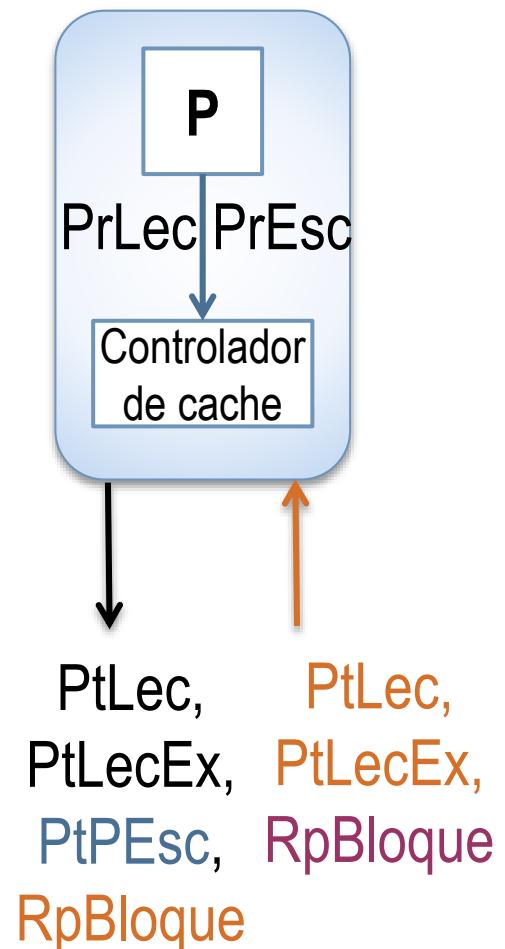
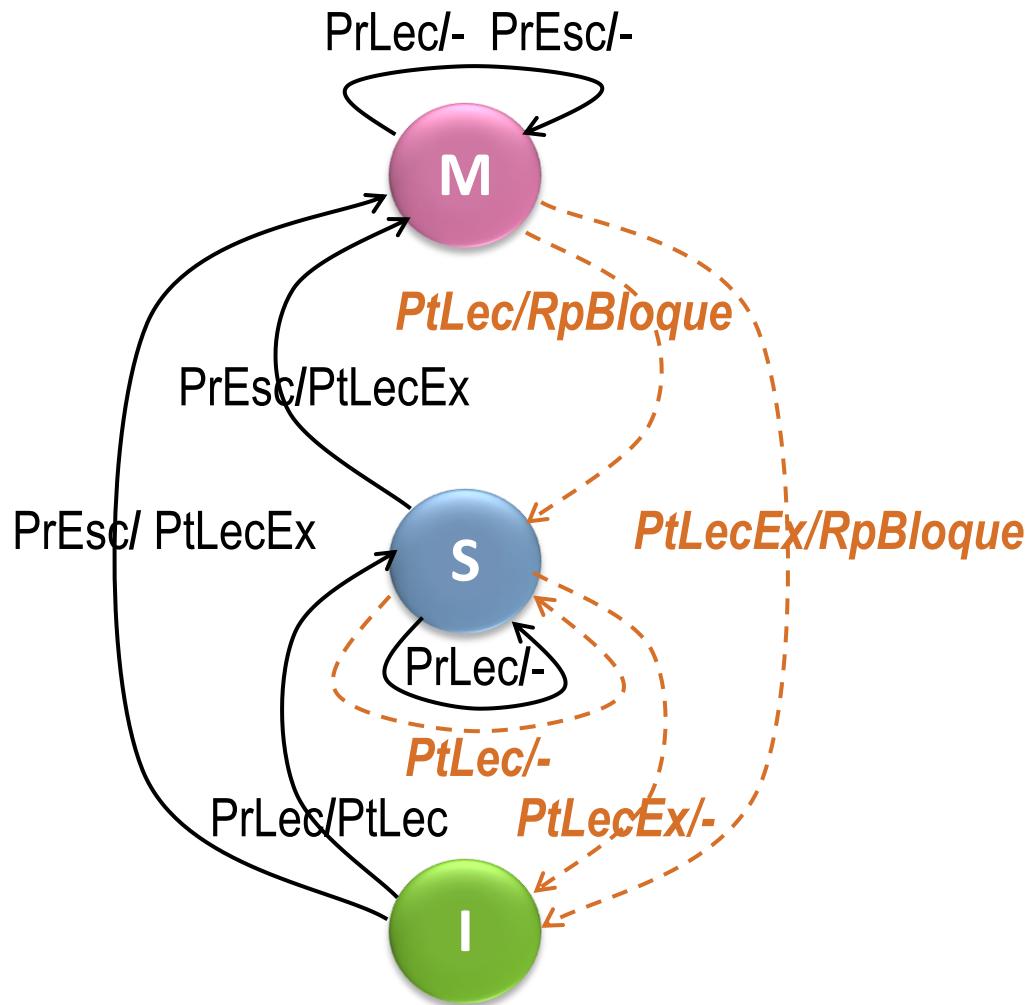
# Protocolo de espionaje de tres estados (MSI) – posescritura e invalidación

- Transferencias generadas por un nodo con cache (tipos de paquetes):
  - Petición de lectura de un bloque (**PtLec**): por lectura con fallo de cache del procesador del nodo (**PrLec**)
  - Petición de acceso exclusivo (**PtLecEx**): por escritura del procesador (**PrEsc**) en bloque compartido o inválido
  - Petición de posescritura (**PtPEsc**): por el reemplazo del bloque modificado (el procesador del nodo no espera respuesta)
  - Respuesta con bloque (**RpBloque**): al tener en estado modificado el bloque solicitado por una PtLec o PtLecEx recibida



PtLec, PtLecEx,  
PtPEsc,  
RpBloque

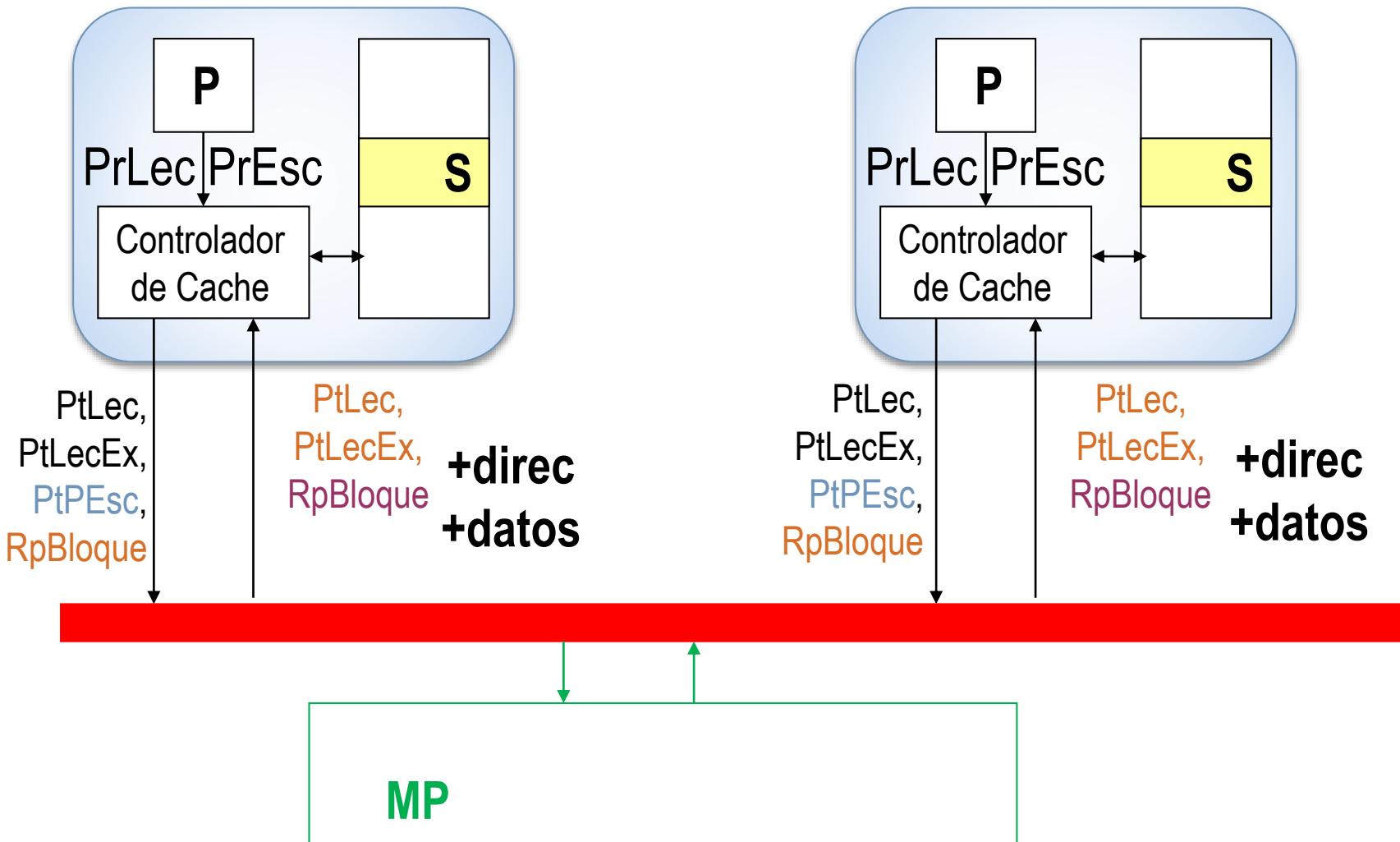
# Diagrama MSI de transiciones de estados



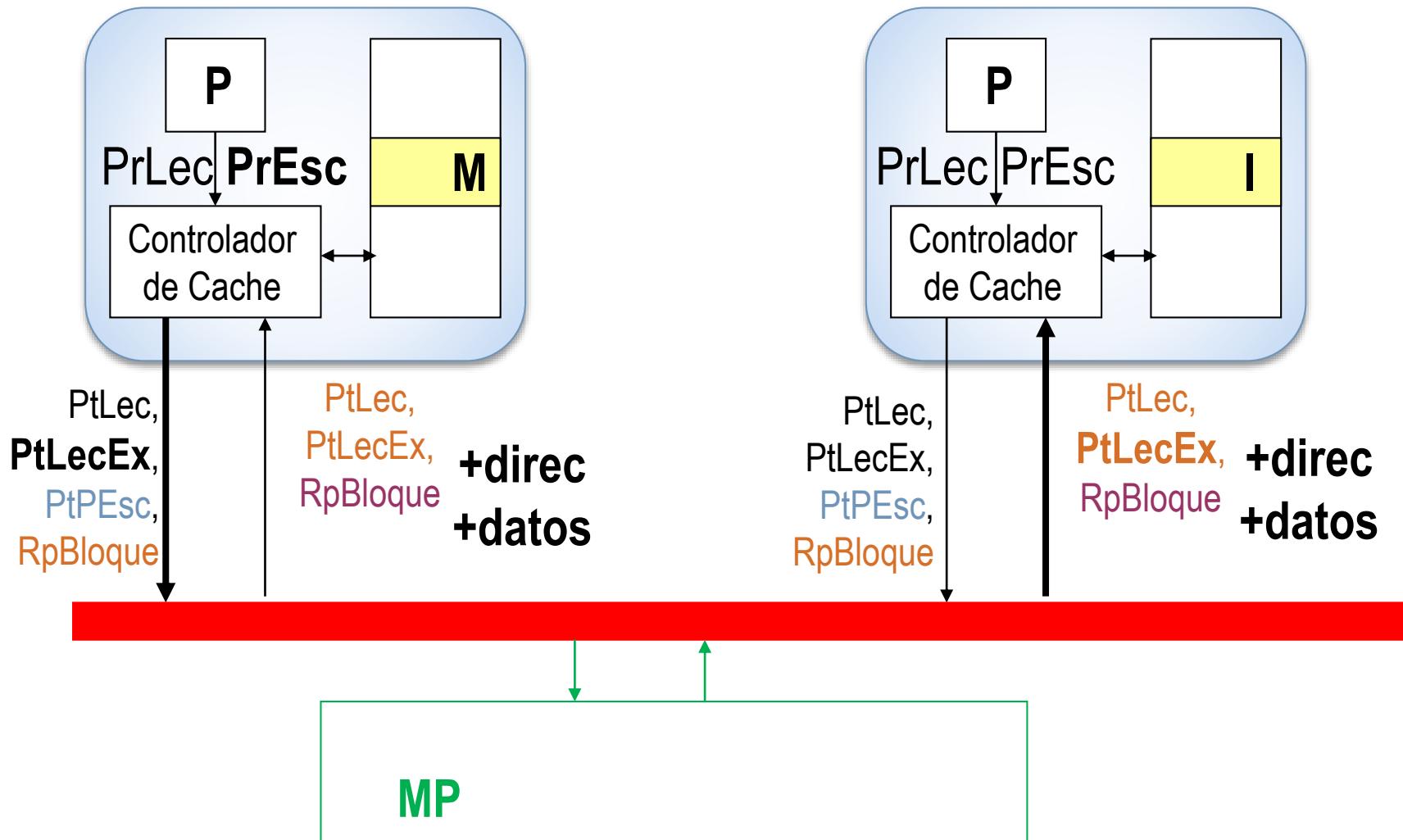
# Tabla de descripción de MSI

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

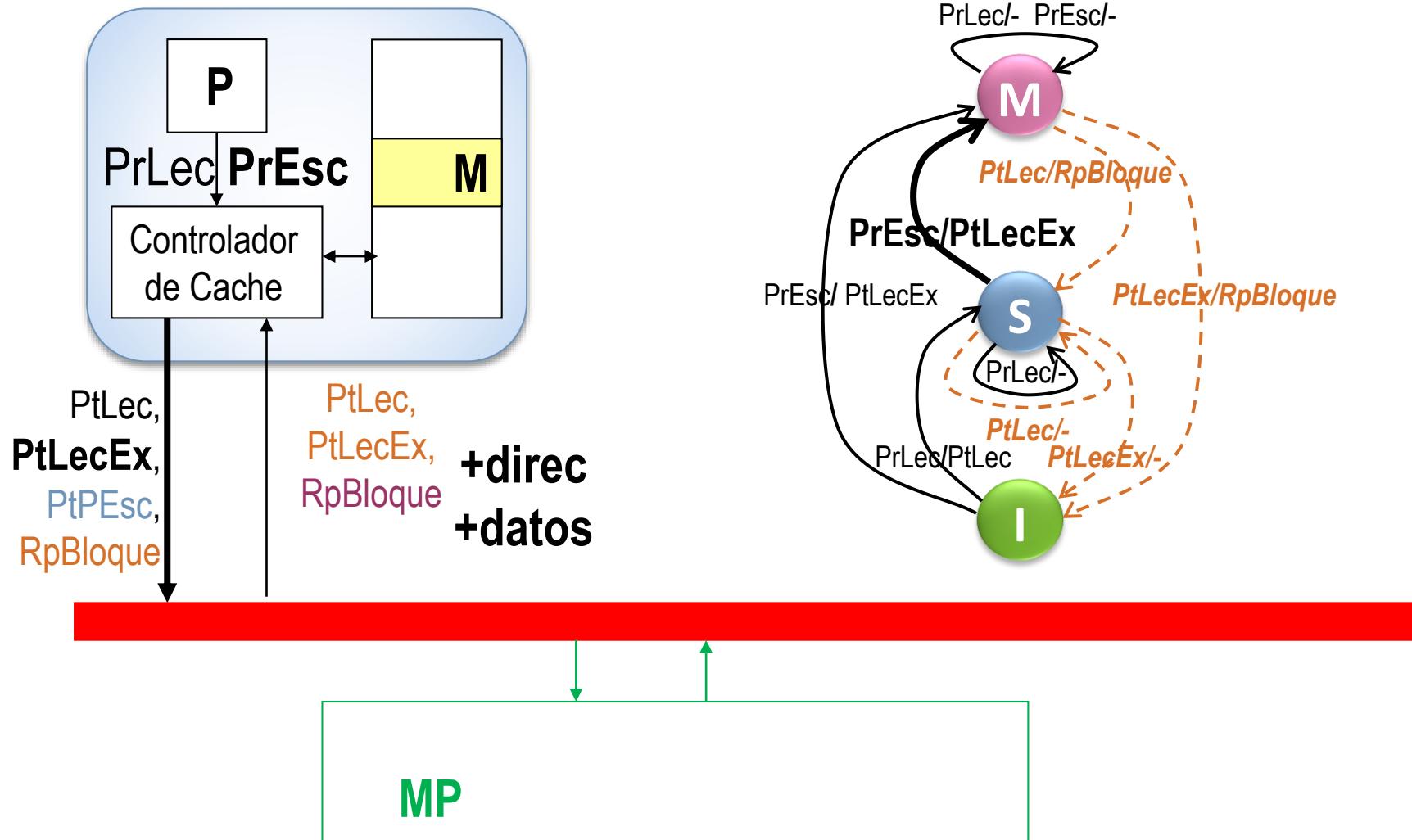
# Ejemplo MSI I



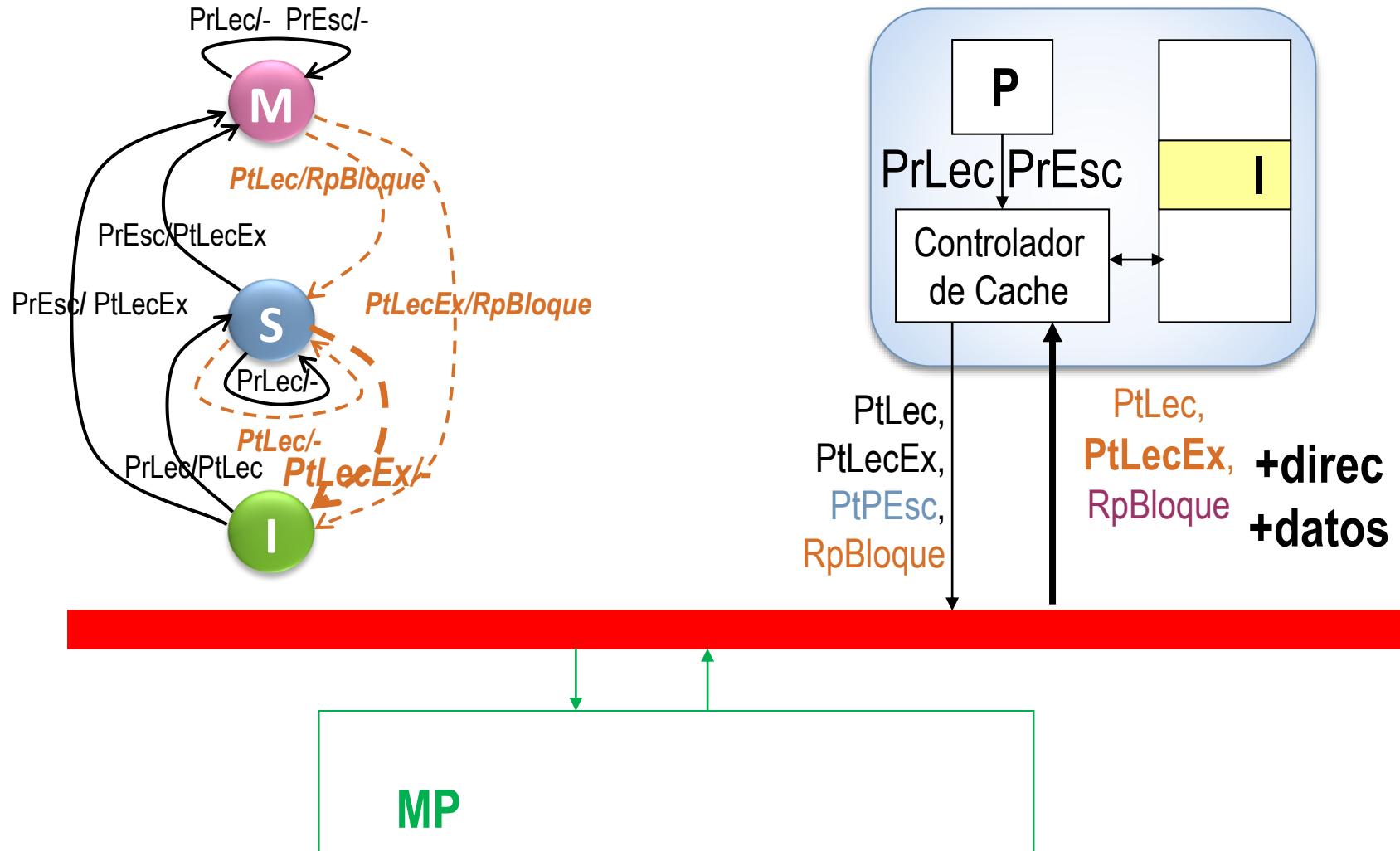
# Ejemplo MSI II



# Ejemplo MSI III



# Ejemplo MSI IV



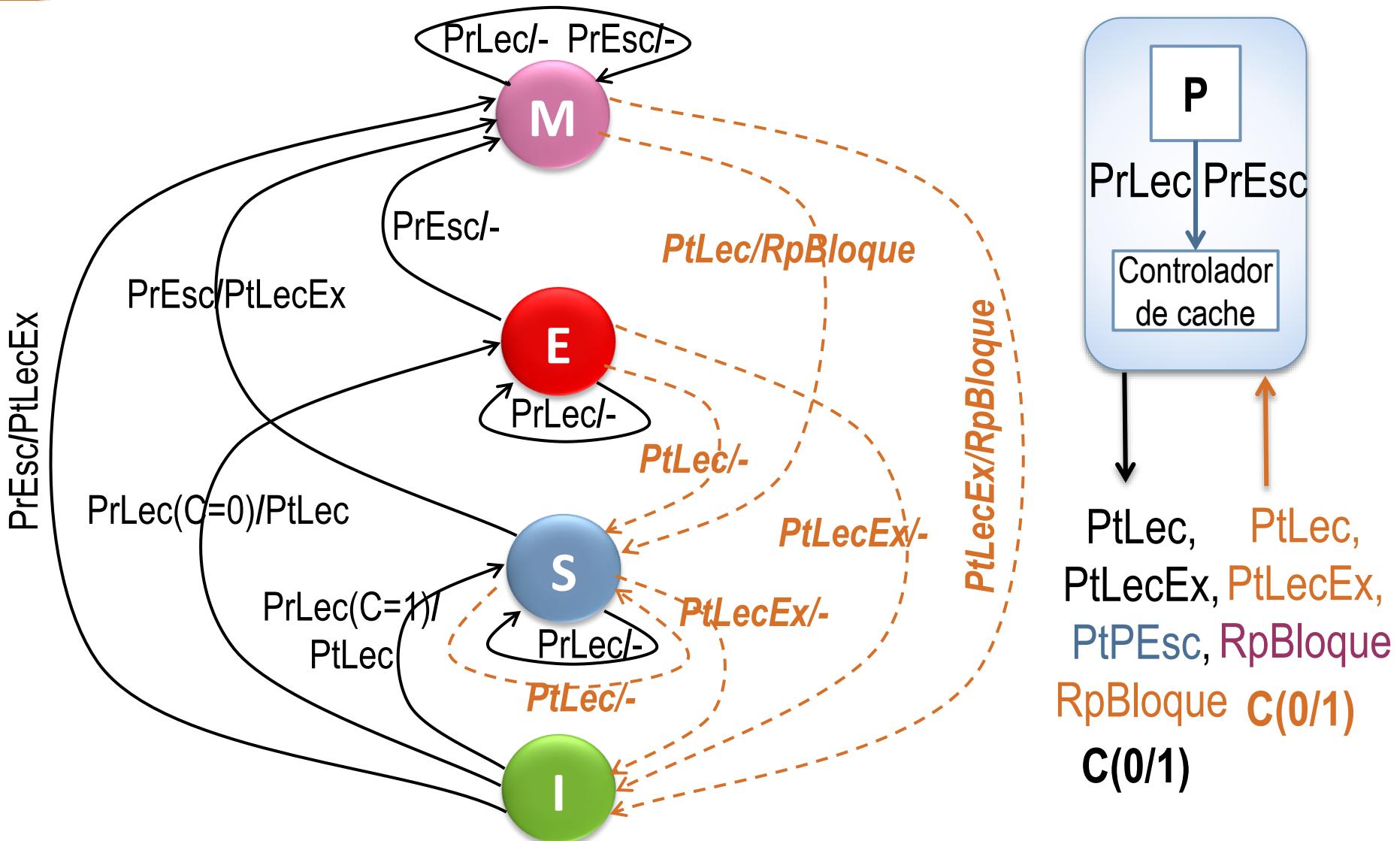
# Contenido Lección 8

- Sistema de memoria en multiprocesadores
- Concepto de coherencia en el sistema de memoria:  
situaciones de incoherencia y requisitos para evitar  
problemas en estos casos
- Protocolos de mantenimiento de coherencia:  
clasificación y diseño
- Protocolo MSI de espionaje
- Protocolo MESI de espionaje
- Protocolo MSI basado en directorios con o sin difusión

# Protocolo de espionaje de cuatro estados (MESI) – posescritura e invalidación

- Estados de un bloque en cache:
  - **Modificado (M)**: es la única copia del bloque válida en todo el sistema
  - **Exclusivo (E)**: es la única copia de bloque válida en caches, la memoria también está actualizada
  - **Compartido (C,Shared)**: es válido, también válido en memoria y en al menos otra cache
  - **Inválido (I)**: se ha invalidado o no está físicamente
- Estados de un bloque en memoria(en realidad se evita almacenar esta información):
  - **Válido**: puede haber copia válida en una o varias caches
  - **Inválido**: habrá copia valida en una cache

# Diagrama MESI de transiciones de estados



# Tabla de descripción de MESI

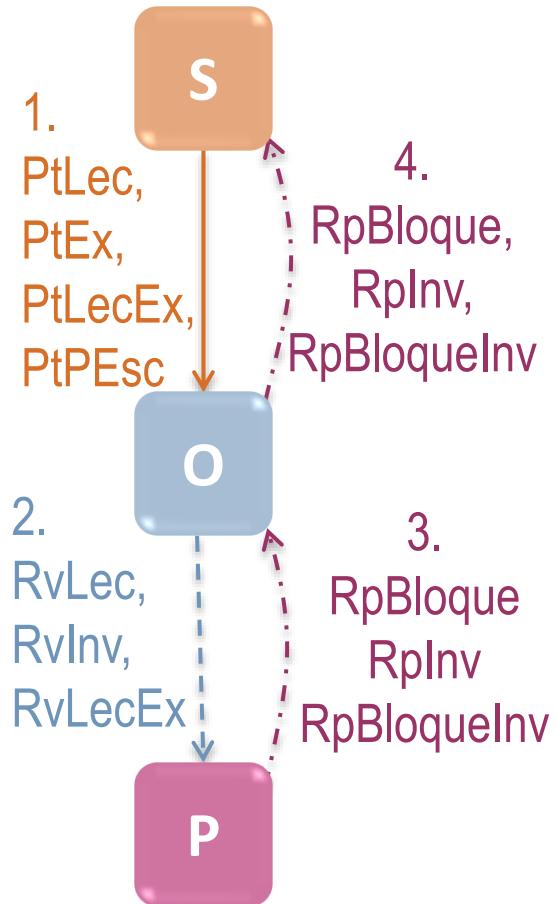
<b>Modificado (M)</b>	PrLec/PrEsc		Modificado
	PtLec	Genera RpBloque	Compartido
	PtLecEx	Genera RpBloque. Invalida copia local	Inválido
	Reemplazo	Genera PtPEsc	Inválido
<b>Exclusivo (E)</b>	PrLec		Exclusivo
	PrEsc		Modificado
	PtLec		Compartido
	PtLecEx	Invalida copia local	Inválido
<b>Compartido (S)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

# Contenido Lección 8

- Sistema de memoria en multiprocesadores
- Concepto de coherencia en el sistema de memoria:  
situaciones de incoherencia y requisitos para evitar  
problemas en estos casos
- Protocolos de mantenimiento de coherencia:  
clasificación y diseño
- Protocolo MSI de espionaje
- Protocolo MESI de espionaje
- Protocolo MSI basado en directorios con o sin difusión

# MSI con directorios (sin difusión) I

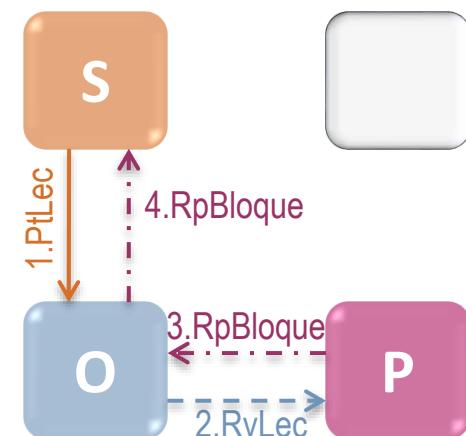
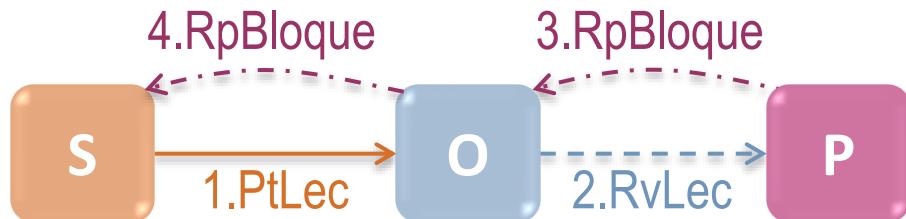
- Estados de un bloque en cache:
  - Modificado (M), Compartido (C), Inválido (I)
- Estados de un bloque en MP:
  - Válido e inválido
- Transferencias (tipos de paquetes) :
  - Tipos de nodos: solicitante (**S**), origen (**O**), modificado (**M**), propietario (**P**) y compartidor (**C**)
  - Petición de
    - nodo **S** a **O**: lectura de un bloque (**PtLec**), lectura con acceso exclusivo (**PtLecEx**), petición de acceso exclusivo sin lectura (**PtEx**), posescritura (**PtPEsc**)
  - Reenvío de petición de
    - nodo **O** a nodos con copia (**P**, **M**, **C**): invalidación (**RvInv**), lectura (**RvLec**, **RvLecEx**).
  - Respuesta de
    - nodo **P** a **O**: respuesta con bloque (**RpBloque**), resp. con o sin bloque confirmando fin inv. (**RpInv**, **RpBloqueInv**)
    - nodo **O** a **S**: resp. con bloque (**RpBloque**), resp. con o sin bloque confirmando fin inv. (**RpInv**, **RpBloqueInv**)



# MSI con directorios (sin difusión) II

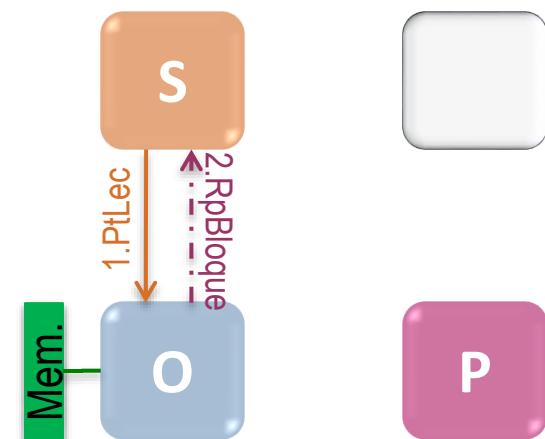
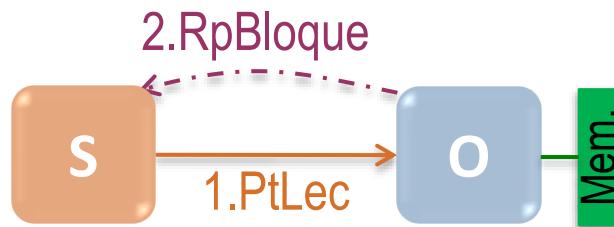
Estado inicial	Evento	Estado final
D) Inválido S) Inválido P) Modificado Acceso remoto	Fallo de lectura	D) Válido S) Compartido P) Compartido

Ejemplo con 4 nodos:



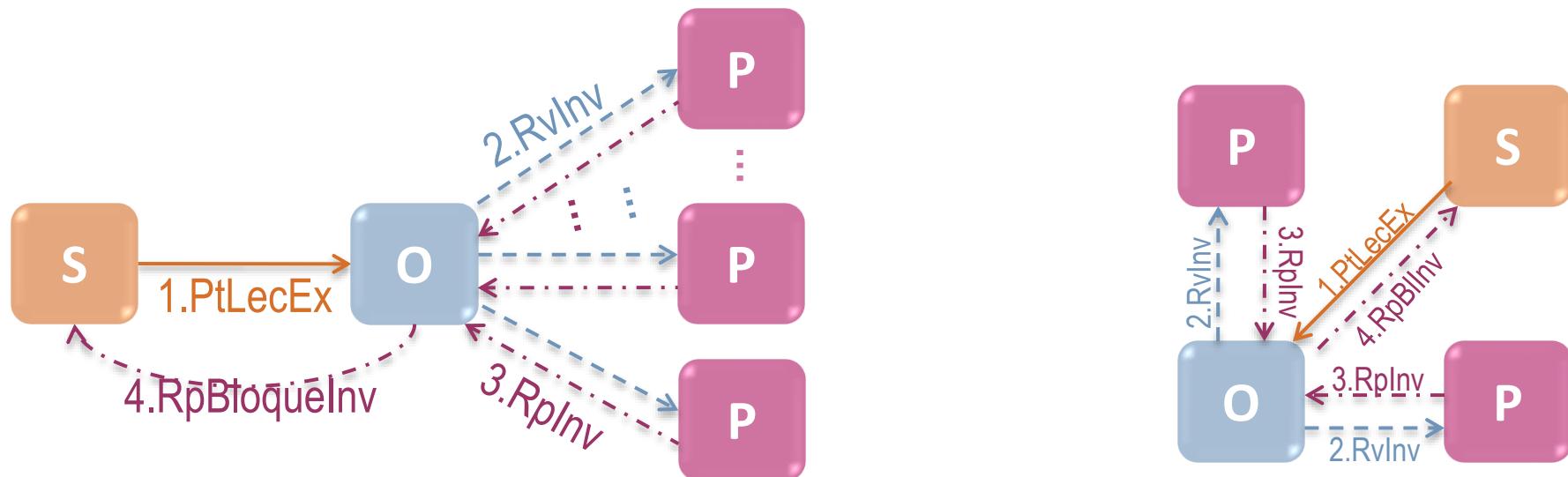
# MSI con directorios (sin difusión) III

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



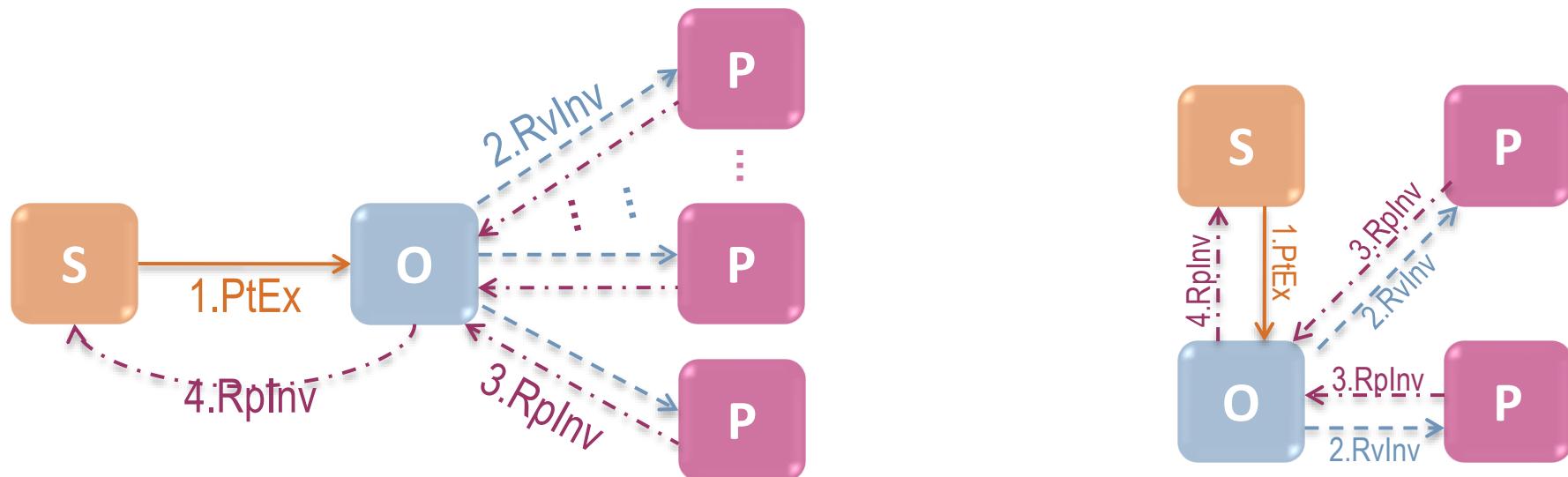
# MSI con directorios (sin difusión) IV

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



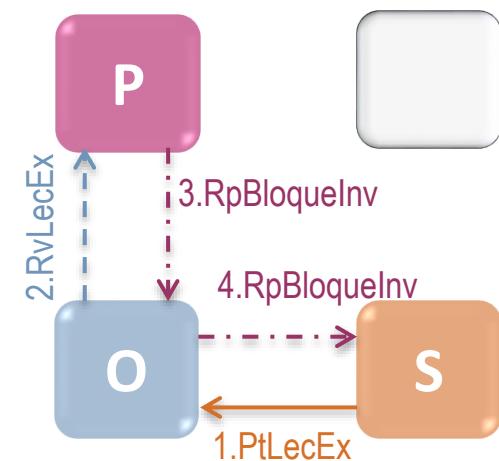
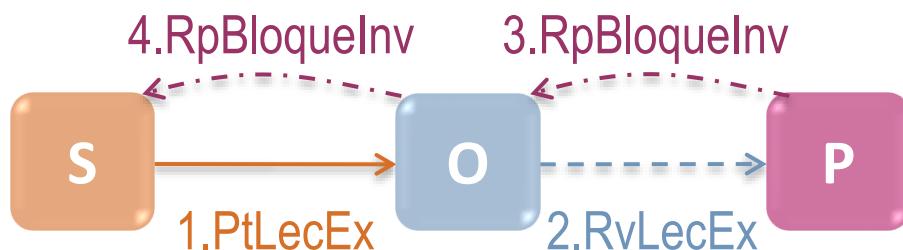
# MSI con directorios (sin difusión) V

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



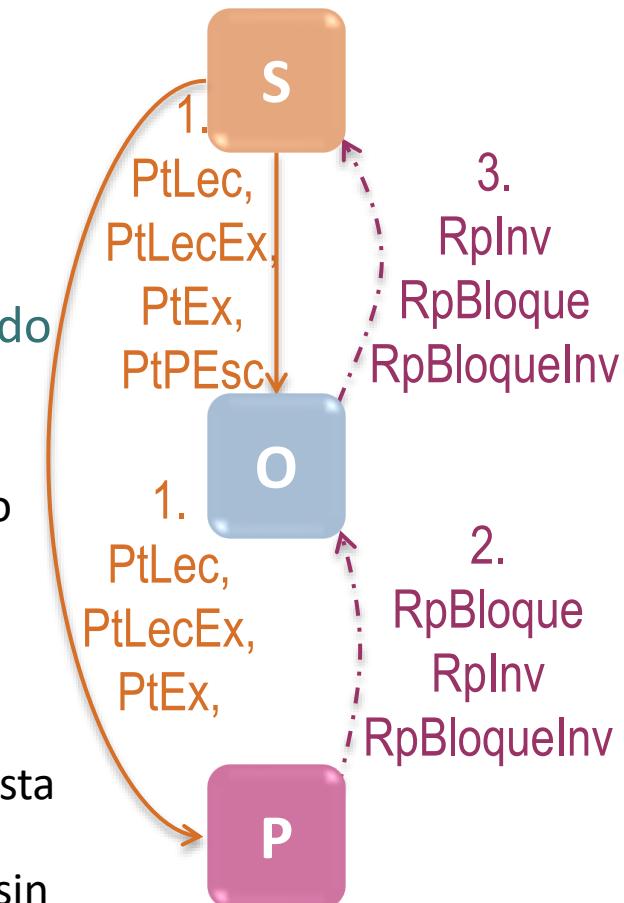
# MSI con directorios (sin difusión) VI

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



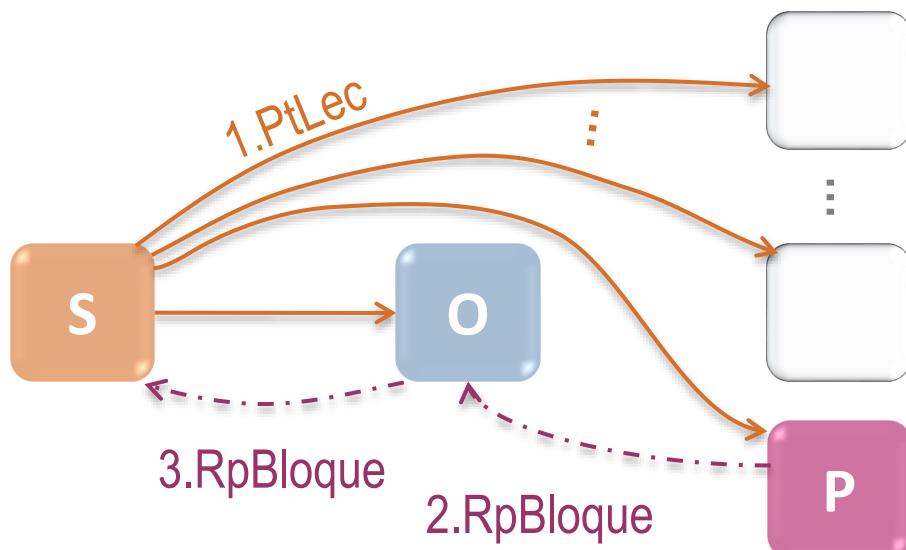
# MSI con directorios (con difusión) I

- Estados de un bloque en cache:
  - Modificado (M), Compartido (C), Inválido (I)
- Estados de un bloque en MP:
  - Válido e inválido
- Transferencias (tipos de paquetes) :
  - Tipos de nodos: solicitante (S), origen (O), modificado (M), propietario (P) y compartidor (C)
  - Difusión de petición del nodo S a
    - O y P: lectura de un bloque (**PtLec**), lectura con acceso exclusivo (**PtLecEx**), petición de acceso exclusivo sin lectura (**PtEx**)
    - O: posescritura (**PtPEsc**)
  - Respuesta de
    - nodo P a O: respuesta con bloque (**RpBloque**), respuesta confirmando invalidación (**Rplnv**)
    - nodo O a S: resp. con bloque (**RpBloque**), resp. con o sin bloque confirmando fin inv. (**Rplnv, RpBloquelnv**)

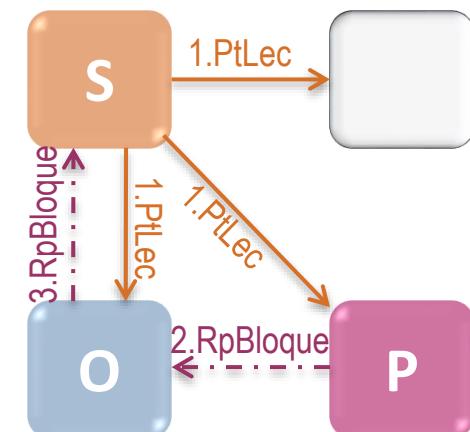


# MSI con directorios (con difusión) II

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

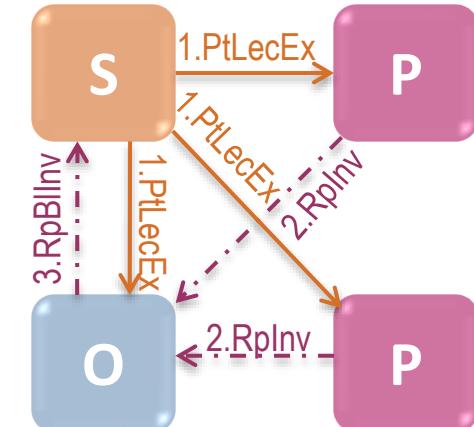
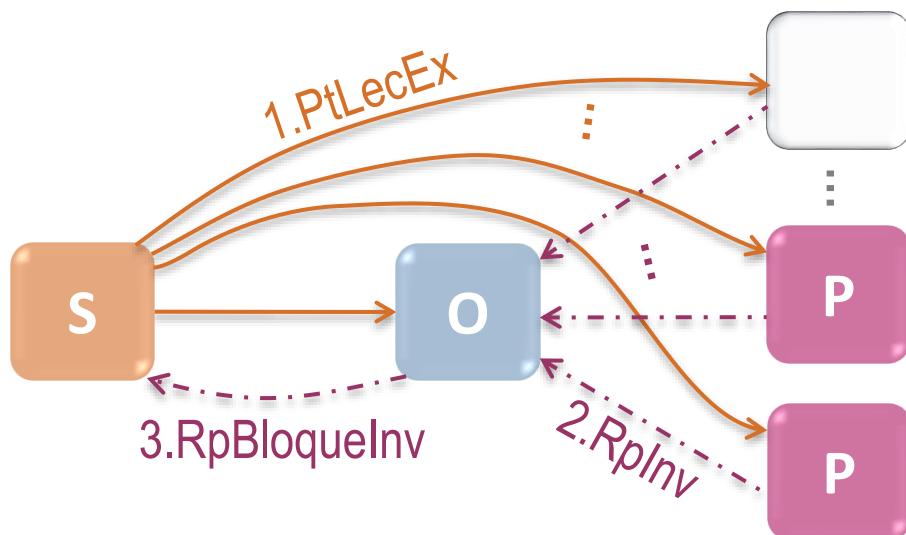


Ejemplo con 4 nodos:



# MSI con directorios (con difusión) III

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



# Para ampliar ...

## ➤ Webs

- An Introduction to the Intel® QuickPath Interconnect,  
<http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html>
- Demo Intel® QuickPath Interconnect  
<http://www.intel.com/content/www/us/en/performance/performance-quickpath-architecture-demo.html>
- Animaciones de protocolos de coherencia de cachés  
<http://lorca.act.uji.es/projects/ccp/>

2º curso / 2º cuatr.

Grados sen  
Ing. Informática

# Arquitectura de Computadores

## Tema 3

### Lección 9. Consistencia del sistema de memoria

Material elaborado por Mancia Anguita y Julio Ortega  
Profesores: Mancia Anguita, Maribel García y Christian Morillas



*ugr*

Universidad  
de Granada



# Lecciones

- Lección 7. Arquitecturas TLP
- Lección 8. Coherencia del sistema de memoria
- Lección 9. Consistencia del sistema de memoria
  - Concepto de consistencia de memoria
  - Consistencia secuencial
  - Modelos de consistencia relajados
- Lección 10. Sincronización

# Objetivos Lección 9

- 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.

# Bibliografía Lección 9

## ➤ Fundamental

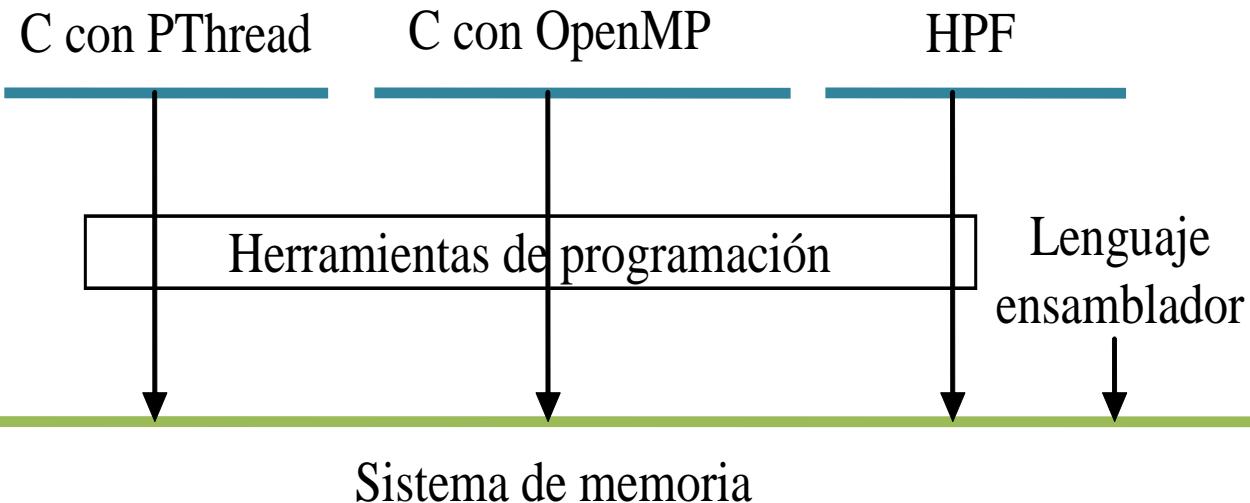
- M. Anguita; J. Ortega. “Fundamentos y Problemas de Arquitectura de Computadores”. Ed. Avicam, 2016.
- Secc. 10.2. J. Ortega, M. Anguita, A. Prieto. “Arquitectura de Computadores”. ESII/C.1 ORT arq

# Contenido Lección 9

- Concepto de consistencia de memoria
- Consistencia secuencial
- Modelos de consistencia relajados

# Consistencia de memoria

Modelo de consistencia  
de memoria Software



Modelo de consistencia  
de memoria Hardware

- Especifica (las restricciones en) **el orden** en el cual las **operaciones de memoria** (lectura, escritura) deben **parecer** haberse realizado (operaciones a las mismas o distintas direcciones y emitidas por el mismo o distinto proceso/procesador)
- La coherencia sólo abarca operaciones realizadas por múltiples componentes (proceso/procesador) en una misma dirección

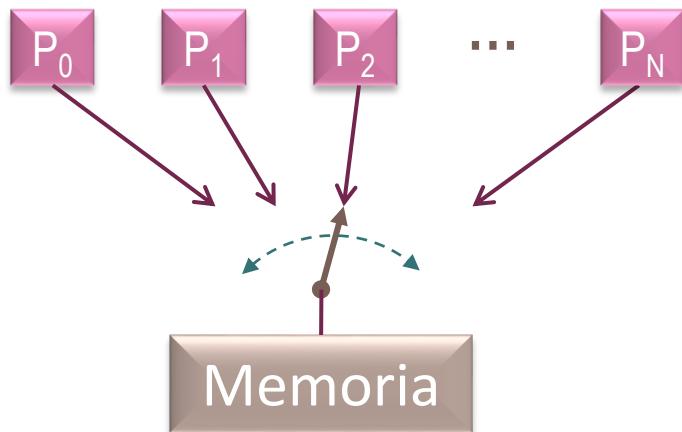
# Contenido Lección 9

- Concepto de consistencia de memoria
- Consistencia secuencial
- Modelos de consistencia relajados

# Consistencia secuencial (SC)

- SC es el modelo de consistencia que espera el programador de las herramientas de alto nivel
- SC requiere que:
  - Todas las operaciones de un único procesador (thread) parezcan ejecutarse en el orden descrito por el programa de entrada al procesador (**orden del programa**)
  - Todas las operaciones de memoria parezcan ser ejecutadas una cada vez (**ejecución atómica**) -> serialización global

# Consistencia Secuencial



- SC presenta el sistema de memoria a los programadores como una **memoria global** conectada a todos los procesadores a través un **comutador central**

# Consistencia Secuencial

Inicialmente  $k1=k2=0$

**P1**

```
k1=1;  
if (k2=0) {  
    Sección crítica  
};
```

**P2**

```
k2=1;  
if (k1=0) {  
    Sección crítica  
};
```

1

¿Qué espera el programador?

**P1**

```
A=1;
```

Inicialmente

**P2**  
if ( $A=1$ )  
 $B=1;$

**P3**

```
if ( $B=1$ )  
reg1=A;
```

2

¿Qué espera el programador que se almacene en reg1 si llega a ejecutarse  $reg1=A$ ?

Inicialmente  $A= 0$

**P1**

```
A=1;  
k=1;
```

**P2**

```
while ( $k=0$ ) {};  
copia=A;
```

3

¿Qué espera el programador que se almacene en copia?

# Ejemplo de Consistencia Secuencial

(1) Escribir K1=1  
(2) Leer K2 ( $\{K2==0\}$ )

1



(a) Escribir K2=1  
(b) Leer K1 ( $\{K1==0\}$ )

Orden con Consistencia Secuencial:

- |              |              |
|--------------|--------------|
| (1)(2)(a)(b) | (a)(b)(1)(2) |
| (1)(a)(2)(b) | (a)(1)(b)(2) |
| (1)(a)(b)(2) | (a)(1)(2)(b) |

(1) Escribir A=1  
(2) Escribir K=1

3



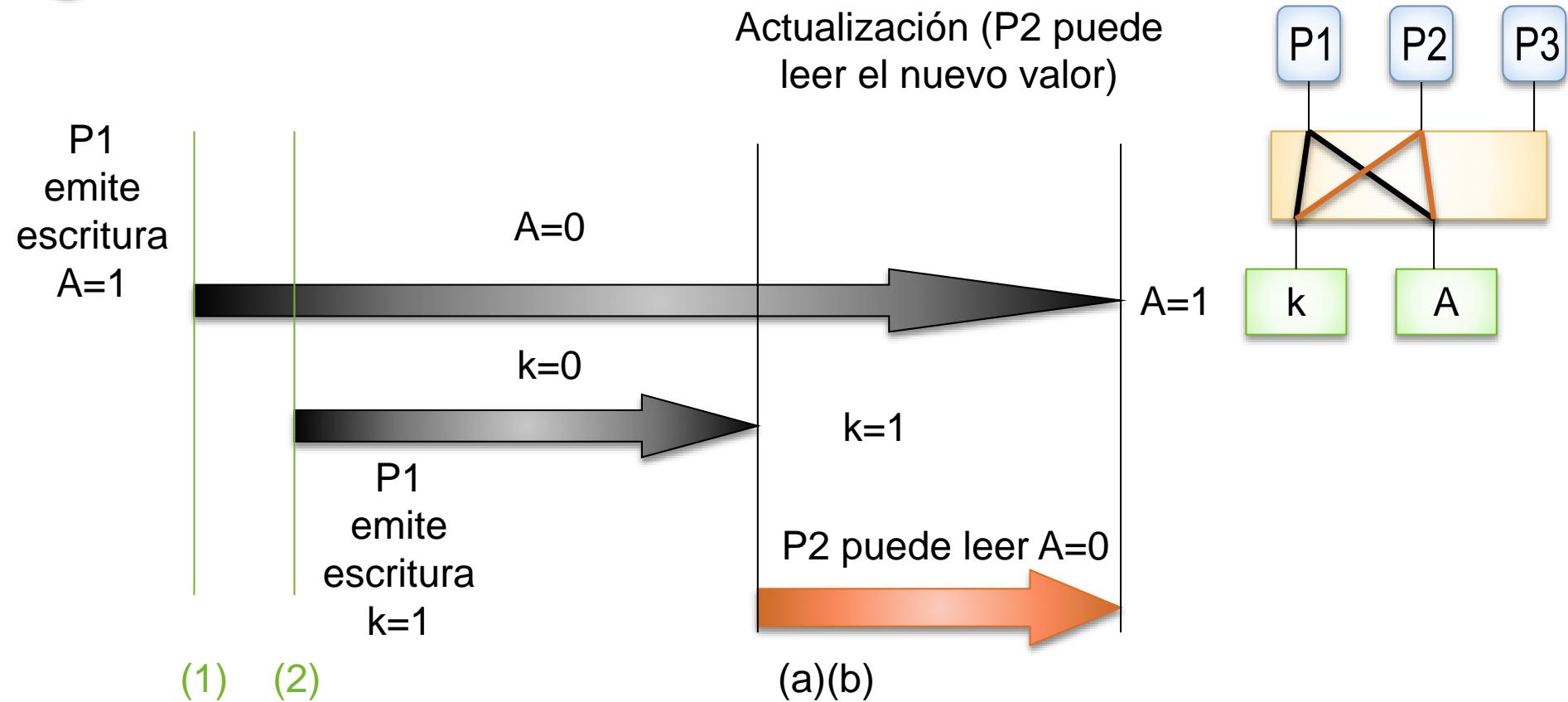
(a) Leer K (while  $k==0 \{\}$ )  
(b) Leer A

Orden con Consistencia Secuencial:

- |                              |
|------------------------------|
| (1)(a)....(a)(2)(a)(b)       |
| (1)(2)(a)(b)                 |
| (a)..(a)(1)(a)..(a)(2)(a)(b) |
| (a)..(a)(1)(2)(a)(b)         |

# ¿Qué puede ocurrir en el computador?

3



No se garantiza el orden  $W \rightarrow W$

# Contenido Lección 9

- Concepto de consistencia de memoria
- Consistencia secuencial
- Modelos de consistencia relajados

# Modelos de consistencia relajados

- Difieren en cuanto a los requisitos para garantizar SC que relajan (los relajan para incrementar prestaciones):
  - Orden del programa:
    - Hay modelos que permiten que se relaje en el código ejecutado en un procesador el orden entre dos acceso a distintas direcciones ( $W \rightarrow R$ ,  $W \rightarrow W$ ,  $R \rightarrow RW$ )
  - Atomicidad:
    - Hay modelos que permiten que un procesador pueda ver el valor escrito por otro antes de que este valor sea visible al resto de los procesadores del sistema
- Los modelos relajados comprenden:
  - Los órdenes de acceso a memoria que no garantiza el sistema de memoria (tanto órdenes de un mismo procesador como atomicidad en las escrituras).
  - Mecanismos que ofrece el hardware para garantizar un orden cuando sea necesario.

# Ejemplos de modelos de consistencia hardware relajados

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

Sigue la tabla de Adve y Gharachorloo en (biblioteca ugr) <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=546611&isnumber=11956>

# Consistencia secuencial

Inicialmente  $k1=k2=0$

**P1**  
 $k1=1;$   
if ( $k2=0$ ) {  
    Sección crítica  
};

**P2**  
 $k2=1;$   
if ( $k1=0$ ) {  
    Sección crítica  
};

①  
NO se comporta como SC los que relajan el orden  $W \rightarrow R$

**P1**  
 $A=1;$   
**P2**  
if ( $A=1$ )  
 $B=1;$

**P3**  
if ( $B=1$ )  
 $reg1=A;$

②  
NO se comporta como SC los que no garantizan atomicidad

Inicialmente  $A= 0$

**P1**  
 $A=1;$   
 $k=1;$

**P2**  
while ( $k=0$ ) {};  
 $copia=A;$

③  
NO se comporta como SC los que relajan el orden  $W \rightarrow W \circ R \rightarrow R$

# Para ampliar ...

## ➤ Artículos en revistas

- Adve, S.V.; Gharachorloo, K.; , "Shared memory consistency models: a tutorial," *Computer* , vol.29, no.12, pp.66-76, Dec 1996. Disponible en (biblioteca ugr):  
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=546611&isnumber=11956>

2º curso / 2º cuatr.

Grados sen  
Ing. Informática

# Arquitectura de Computadores

## Tema 3

### Lección 10. Sincronización

Material elaborado por Mancia Anguita y Julio Ortega

Profesores: Mancia Anguita, Maribel García y Christian Morillas



*ugr*

Universidad  
de Granada



# Lecciones

- Lección 7. Arquitecturas TLP
- Lección 8. Coherencia del sistema de memoria
- Lección 9. Consistencia del sistema de memoria
- Lección 10. Sincronización
  - Comunicación en multiprocesadores y necesidad de usar código de sincronización
  - Soporte software y hardware para sincronización
  - Cerrojos
    - Cerrojos simples
    - Cerrojos con etiqueta
  - Barreras
  - Apoyo hardware a primitivas software

# Objetivos Lección 10

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

# Bibliografía Lección 10

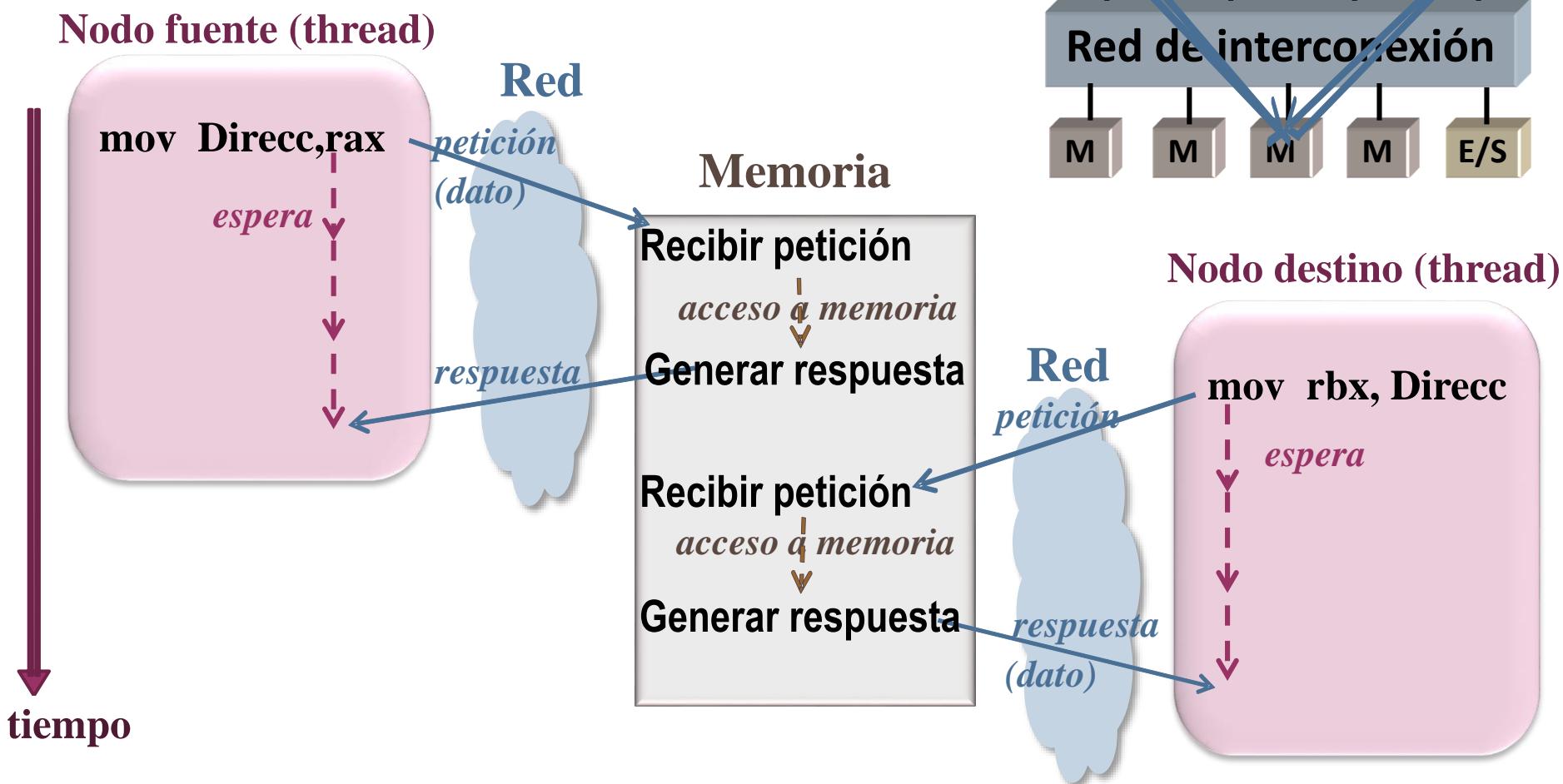
## ➤ Fundamental

- M. Anguita, J. Ortega: "Fundamentos y Problemas de Arquitectura de Computadores". Ed. Avicam, 2016.
- Secc. 10.3. J. Ortega, M. Anguita, A. Prieto. "Arquitectura de Computadores". ESII/C.1 ORT arq

# Contenido Lección 10

- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software

# Comunicación en un multiprocesador



# Comunicación uno-a-uno

Secuencial	Paralela	
	<u>P1</u>	<u>P2</u>
... <b>A=valor;</b> ... <b>copia=A;</b> ...	... <b>A=valor;</b> ...	... <b>copia=A;</b> ...
... <b>mov A,rx</b> ... <b>mov rbx,A</b> ...	... <b>mov A,rx</b> ...	... <b>mov rbx,A</b> ...

# Comunicación uno-a-uno. Necesidad de sincronización

- Se debe garantizar que el proceso que recibe lea la variable compartida cuando el proceso que envía haya escrito en la variable el dato a enviar
- Si se reutiliza la variable para comunicación, se debe garantizar que no se envía un nuevo dato en la variable hasta que no se haya leído el anterior

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

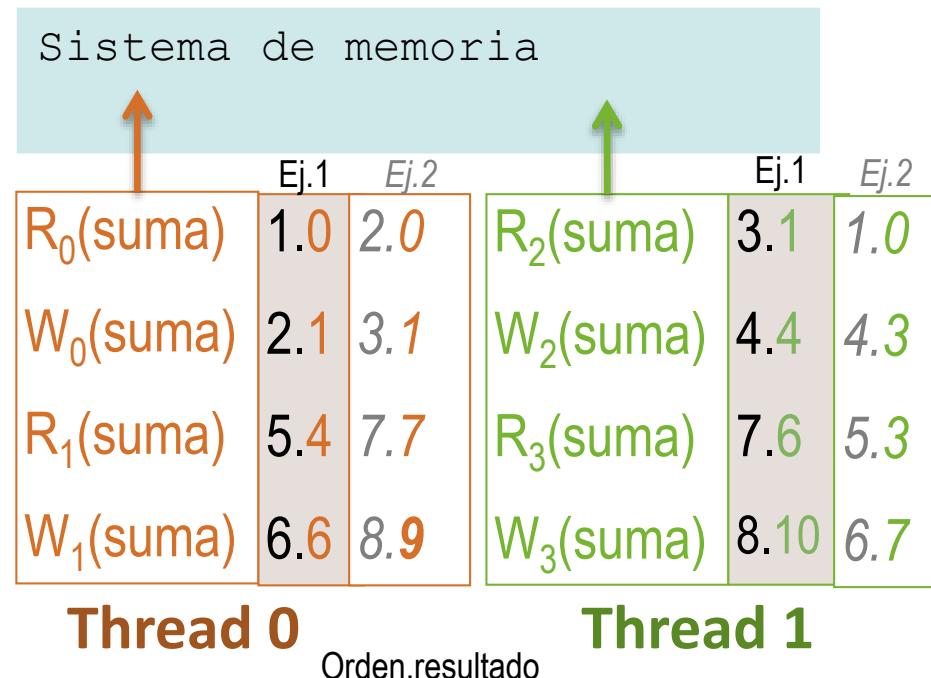
# Comunicación colectiva

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

## ➤ Ejemplo de comunicación colectiva: suma de n números:

- La lectura-modificación-escritura de `sum` se debería hacer en exclusión mutua (es una sección crítica) => **cerrojos**
  - Sección crítica: Secuencia de instrucciones con una o varias direcciones compartidas (variables) que se deben acceder en exclusión mutua
- El proceso 0 no debería imprimir hasta que no hayan acumulado `sump` en `sum` todos los procesos => **barreras**

# Comunicación colectiva en multiprocesadores (carrera)



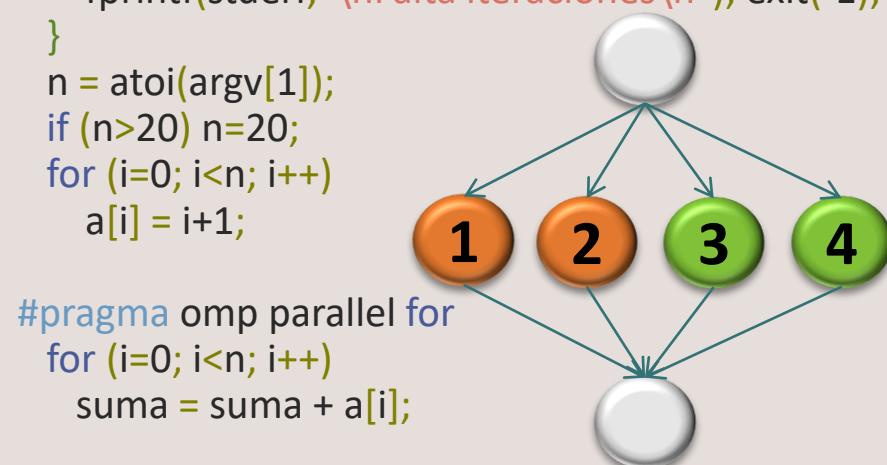
- Ej. para n=4, el compilador no optimiza
  - $a=\{1,2,3,4\}$
  - $R_i$  (suma): Lectura de suma en la iteración i

**sin exclusión mutua en el acceso a suma**

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
main(int argc, char **argv) {
    int i, n=20, a[n], suma=0;
    if(argc < 2) {
        fprintf(stderr, "\nFalta iteraciones\n"); exit(-1);
    }
    n = atoi(argv[1]);
    if (n>20) n=20;
    for (i=0; i<n; i++)
        a[i] = i+1;
```



```
graph TD; Root(( )) --> Node1((1)); Root --> Node2((2)); Root --> Node3((3)); Root --> Node4((4))
```

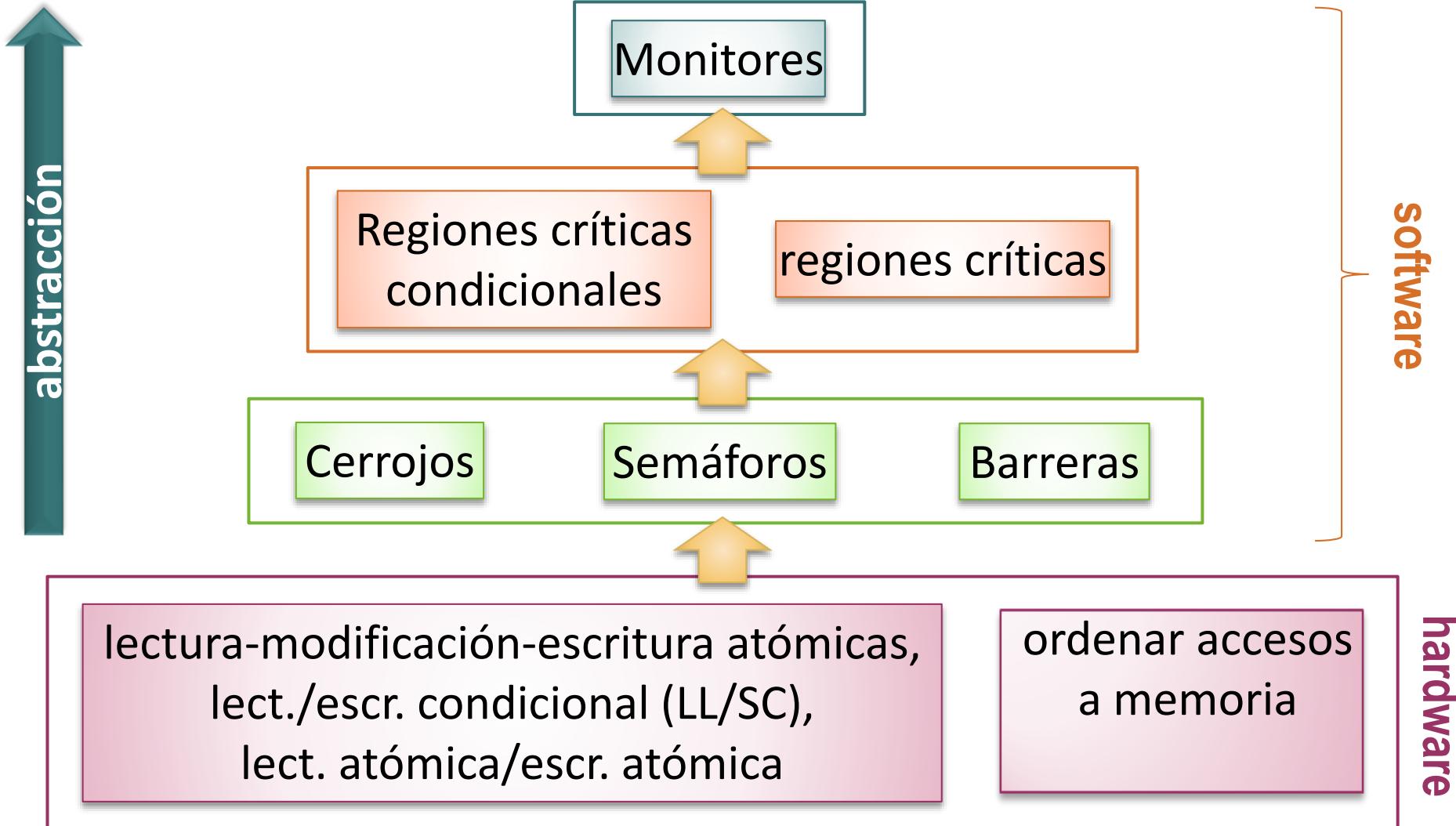


```
#pragma omp parallel for  
for (i=0; i<n; i++)  
    suma = suma + a[i];  
  
printf("Fuera de 'parallel' suma=%d\n",suma);  
return(0);  
}
```

# Contenido Lección 10

- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software

# Soporte software y hardware de sincronización

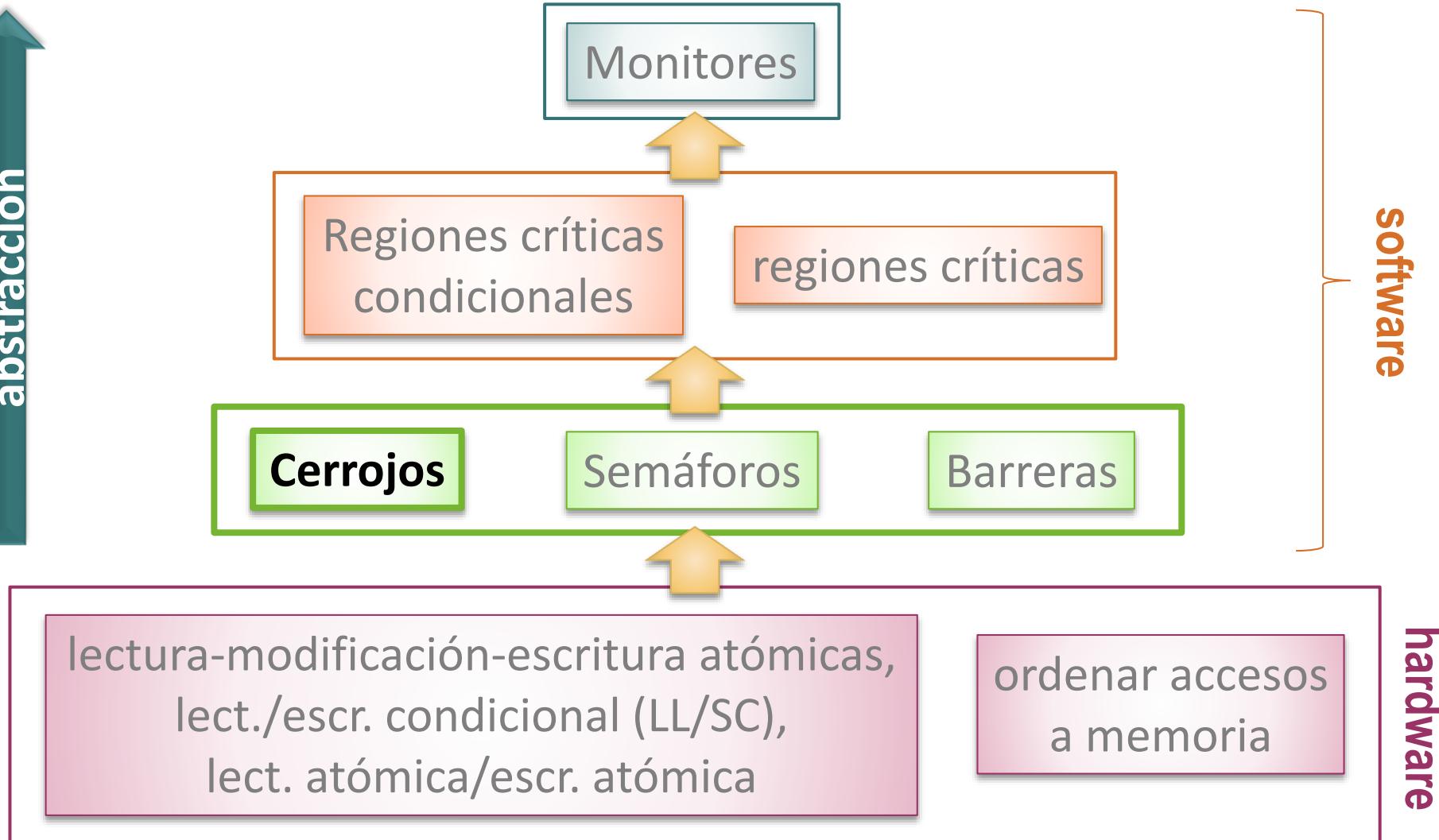


# Contenido Lección 10

- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
  - Cerrojos simples
  - Cerrojos con etiqueta
- Barreras
- Apoyo hardware a primitivas software

# Soporte software y hardware de sincronización

abstracción



# Cerrojos

- Permiten sincronizar mediante dos operaciones:
  - **Cierre del cerrojo o lock (k)** : intenta **adquirir** el derecho a acceder a una sección crítica (cerrando o adquiriendo el cerrojo k).
    - Si varios procesos intentan la **adquisición** (cierre) a la vez, sólo uno de ellos lo debe conseguir, el resto debe pasar a una *etapa de espera*.
    - Todos los procesos que ejecuten `lock ()` con el cerrojo cerrado deben quedar **esperando**.
  - **Apertura del cerrojo o unlock (k)** : **libera** a uno de los threads que esperan el acceso a una sección crítica (éste adquiere el cerrojo).
    - Si no hay threads en **espera**, permitirá que el siguiente thread que ejecute la función `lock ()` adquiera el cerrojo k sin espera.

# Cerrojos en ejemplo suma

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

- Alternativas para implementar la espera:
  - Espera ocupada.
  - Suspensión del proceso o thread, éste queda esperando en una cola, el procesador conmuta a otro proceso-thread.

# Componentes en un código para sincronización

## ➤ Método de adquisición

- Método por el que un thread trata de adquirir el derecho a pasar a utilizar unas direcciones compartidas. Ej.:
  - Utilizando lectura-modificación-escritura atómicas: x86, Intel Itanium, Sun Sparc
  - Utilizando LL/SC (*Load Linked / Store Conditional*): IBM Power/PowerPC

## ➤ Método de espera

- Método por el que un thread espera a adquirir el derecho a pasar a utilizar unas direcciones compartidas:
  - Espera ocupada (*busy-waiting*)
  - Bloqueo

## ➤ Método de liberación

- Método utilizado por un thread para liberar a uno (cerrojo) o varios (barrera) threads en espera

# Cerrojo Simple I

- Se implementa con una variable compartida  $k$  que toma dos valores: abierto (0), cerrado (1)
- **Apertura** del cerrojo,  $unlock(k)$  : abre el cerrojo escribiendo un 0 (*operación invisible*)
- **Cierre** del cerrojo,  $lock(k)$  : Lee el cerrojo y lo cierra escribiendo un 1.
  - **Resultado de la lectura:**
    - si el cerrojo **estaba cerrado** el thread espera hasta que otro thread ejecute  $unlock(k)$ ,
    - si **estaba abierto** adquiere el derecho a pasar a la sección crítica.
  - **leer-assignar\_1-escibir en el cerrojo debe ser invisible (atómica)**

# Cerrojo Simple II

- Se debe añadir lo necesario para garantizar el acceso en exclusión mutua a k y el orden imprescindible en los accesos a memoria

## lock (k)

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

## unlock (k)

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

# Cerrojos en OpenMP

Descripción	Función de la biblioteca OpenMP
Iniciar (estado unlock)	omp_init_lock (&k)
Destruir un cerrojo	omp_destroy_lock (&k)
Cerrar el cerrojo lock (k)	omp_set_lock (&k)
Abrir el cerrojo unlock (k)	omp_unset_lock (&k)
Cierre del cerrojo pero sin bloqueo (devuelve 1 si estaba cerrado y 0 si está abierto)	omp_test_lock (&k)

# Cerrojos con etiqueta

- Fijan un orden FIFO en la adquisición del cerrojo (se debe añadir lo necesario para garantizar el acceso en exclusión mutua a contadores y el orden imprescindible en los accesos a memoria):

## lock (contadores)

```
contador_local_adq = contadores.adq;  
contadores.adq = (contadores.adq + 1) mod max_flujos_control;  
while (contador_local_adq <> contadores.lib) {};
```

## unlock (contadores)

```
contadores.lib = (contadores.lib + 1) mod max_flujos_control;
```

# Contenido Lección 10

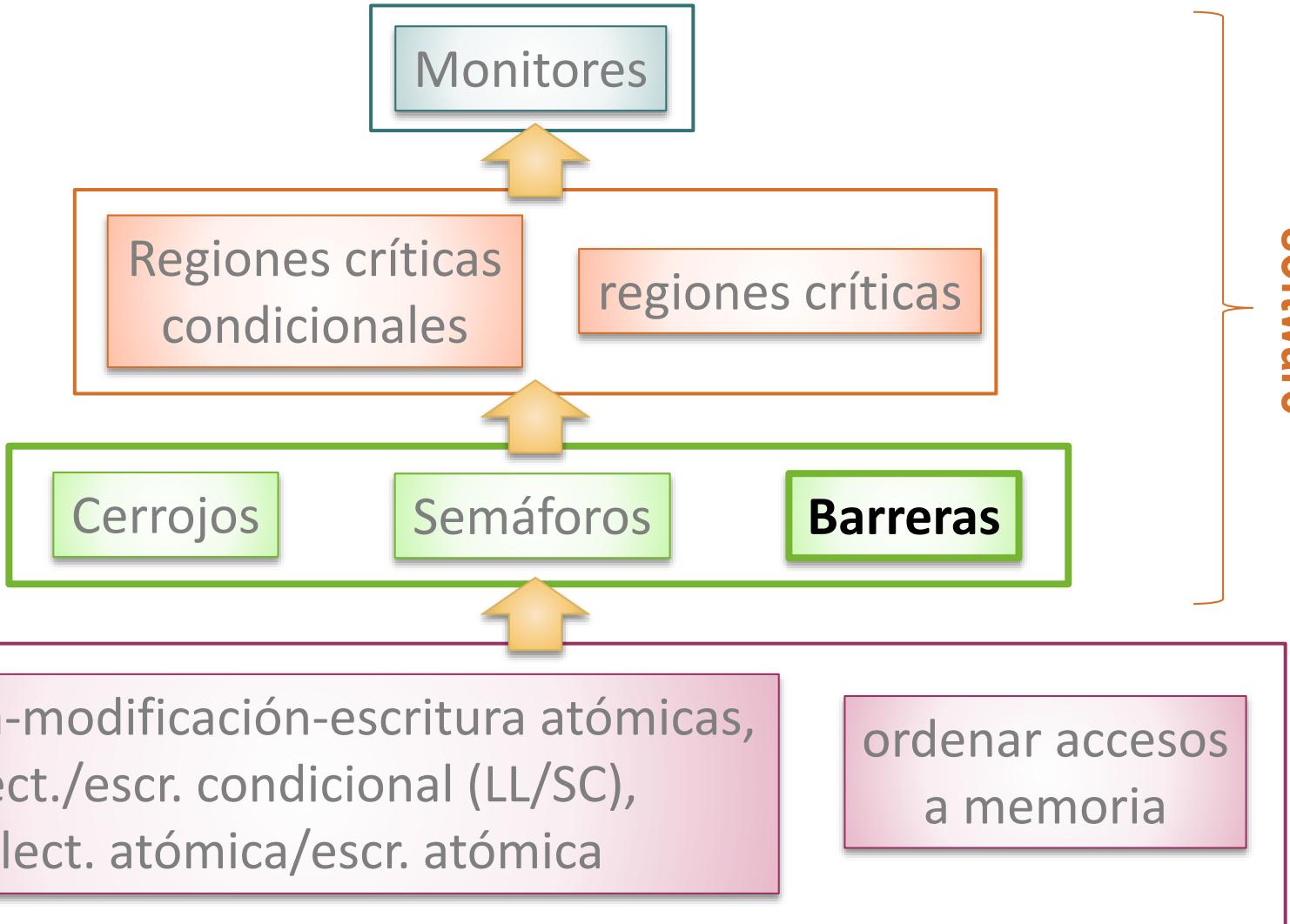
- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software

# Soporte software y hardware de sincronización

abstracción

software

hardware



# Barreras

```
main (){  
    ...  
    Barrera(g,4)  
    ...  
}
```

```
Barrera(id, num_procesos) {  
    if (bar[id].cont==0) bar[id].bandera=0;  
    cont_local = ++bar[id].cont;  
    if (cont_local ==num_procesos) {  
        bar[id].cont=0;  
        bar[id].bandera=1;  
    }  
    else espera mientras bar[id].bandera=0;  
}
```

- Acceso Ex. Mutua.

- Implementar **espera**. Si espera ocupada:  
**while (bar[id].bandera==0) {};**

# Barreras sin problema de reutilización

## 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
```

```
}
```

# Contenido Lección 10

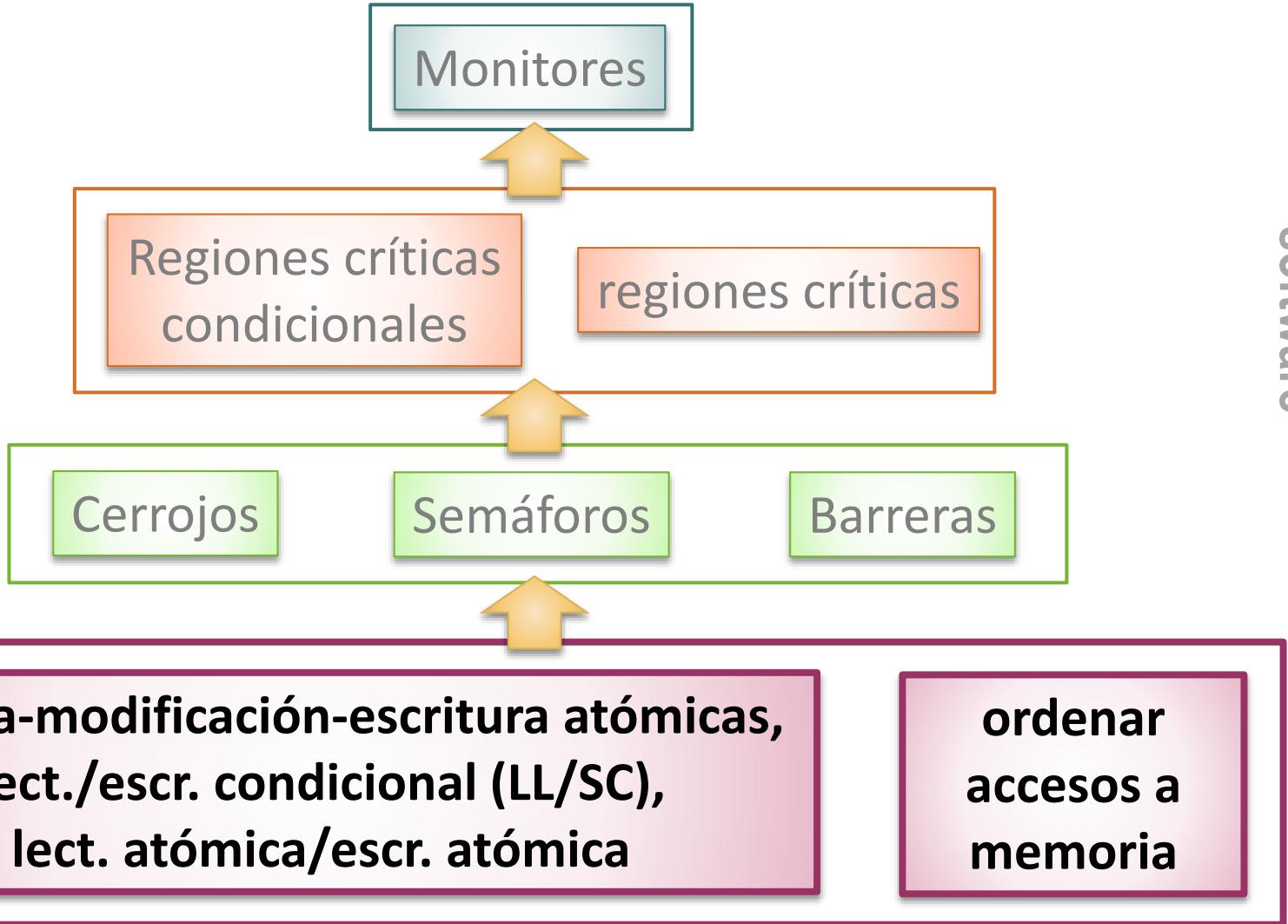
- Comunicación en multiprocesadores y necesidad de usar código de sincronización
- Soporte software y hardware para sincronización
- Cerrojos
- Barreras
- Apoyo hardware a primitivas software
  - Instrucciones de lectura-modificación-escritura atómicas
  - Instrucciones LL/SC (*Load Linked / Store Conditional*)

# Soporte software y hardware de sincronización



software

hardware



# Instrucciones de lectura-modificación-escritura atómicas

## Test&Set (x)

```
Test&Set (x) {  
    temp = x ;  
    x = 1 ;  
    return (temp) ;  
}  
/* x compartida */
```

x86

```
mov reg,1  
xchg reg,mem  
reg ↔ mem
```

## Fetch&Oper(x,a)

```
Fetch&Add(x,a) {  
    temp = x ;  
    x = x + a ;  
    return (temp)  
}/* x compartida,  
a local */
```

x86

```
lock xadd reg,mem  
reg ← mem |  
mem ← reg+mem
```

## Compare&Swap(a,b,x)

```
Compare&Swap(a,b,x){  
    if (a==x) {  
        temp=x ;  
        x=b; b=temp ; }  
}/* x compartida,  
a y b locales */
```

x86

```
lock cmpxchg mem,reg  
if eax=mem  
then mem ← reg  
else eax ← mem
```

# Cerrojos simples con Test&Set y Fetch&Or

## 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
```

## Fetch&Oper(x,a)

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

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

# Cerrojos simples con Compare&Swap

**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; }  
}
```

# Cerrojo simple en Itanium (consistencia de liberación) con Compare&Swap



```
lock:                                //lock(M[lock])
    mov ar.ccv = 0                  // cmpxchg compara con ar.ccv
                                    // que es un registro de propósito específico
    mov r2 = 1                      // cmpxchg utilizará r2 para poner el cerrojo a 1
spin:                                 // se implementa espera ocupada
    ld8 r1 = [lock] ;;             // carga el valor actual del cerrojo en r1
    cmp.eq p1,p0 = r1, r2;        // si r1=r2 entonces cerrojo está a 1 y se hace p1=1
    (p1) br.cond.spnt spin ;;   // si p1=1 se repite el ciclo; spnt, indica que se
                                // usa una predicción estática para el salto de "no tomar"
cmpxchg8.acq r1 = [lock], r2 ;; //intento de adquisición escribiendo 1
                                // IF [lock]=ar.ccv THEN [lock]←r2; siempre r1←[lock]
    cmp.eq p1, p0 = r1, r2       // si r1!=r2 (r1=0) => cer. era 0 y se hace p1=0
    (p1) br.cond.spnt spin ;; // si p1=1 se ejecuta el salto
```

```
unlock:                                //unlock(M[lock])
st8.rel [lock] = r0 ;; //liberar asignando un 0, en Itanium r0 siempre es 0
```

# Cerrojo simple en Itanium (consistencia de liberación) con Compare&Swap

AC ATC

```
lock: //lock(M[lock])
```

```
    mov ar.ccv = 0
```

```
    mov r2 = 1
```

```
spin:
```

```
    ld8 r1 = [lock] ;;
```

```
    cmp.eq p1,p0 = r1, r2;
```

```
    (p1) br.cond.spnt spin ;;
```

```
    cmpxchg8.acq r1 = [lock], r2 ;;
```

```
    cmp.eq p1, p0 = r1, r2
```

```
    (p1) br.cond.spnt spin ;;
```

Si

[lock] != ar.ccv (=0)

r1=1 y p1=1 (r2=1)

Si [lock]==1

(si ya es 1 no hay que utilizar  
instrucción atómica)

Si

[lock]==ar.ccv (=0)

r1=0 y p1=0 (r2=1)

```
unlock: //unlock(M[lock])
```

```
    st8.rel [lock] = r0 ;; //liberar asignando un 0, en Itanium r0 siempre es 0
```

# Cerrojo simple en PowerPC(consistencia débil) con LL/SC implementando Test&Set



```
lock:          #lock(M[r3])
    li      r4,1  #para cerrar el cerrojo
bucle:  lwarx  r5,0,r3 #carga y reserva: r5←M[r3] (instrucción LL)
        cmpwi r5,0  #si está cerrado (a 1)
        bne-   bucle #esperar en el bucle (r5==1), en caso contrario (r5=0)
        stwcx. r4,0,r3 #poner a 1 (recordar: r4=1): M[r3] ← r4 (inst. SC)
        bne-   bucle #el thread repite si ha perdido la reserva (marca de r3 es 0)
        isync           #accede a datos compartidos cuando sale del bucle
```

```
unlock:
    sync           # unlock(M[r3])
    li      r1,0
    stw    r1,0(r3) #abre el cerrojo
```

# Algoritmos eficientes con primitivas hardware

## Suma con fetch&add

```
for (i=ithread; i<n; i=i+nthread)
    fetch&add(sum,a[i]);
/* sum variable compartida */
```

## Suma con fetch&add

```
for (i=ithread; i<n; i=i+nthread)
    sump = sump + a[i];
fetch&add(sum,sump);
/* sum variable compartida */
```

## Suma con compare&swap

```
for (i=ithread; i<n; i=i+nthread)
    sump = sump + a[i];
do
    a = sum;
    b = a + sump;
    compare&swap(a,b,sum);
    while (a!=b);
/* sum variable compartida */
```

# Para ampliar ...

## ➤ Webs

- Implementación en el kernel de linux de cerrojos con etiqueta <http://lxr.free-electrons.com/source/arch/x86/include/asm/spinlock.h>

## ➤ Artículos en revistas

- Graunke, G.; Thakkar, S.; , "Synchronization algorithms for shared-memory multiprocessors," *Computer* , vol.23, no.6, pp.60-69, Jun 1990. Disponible en (biblioteca ugr):  
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=55501&isnumber=2005>