Lección 7. Arquitecturas TLP

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

- Multiprocesador.-Ejecutan varios threads en paralelo en un computador con varios cores/procesadores (cada thread en un core/procesador distinto).
- 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 clasificación: sistema de memoria
    - Con memoria centralizada (UMA).- Mayor latencia – Poco escalable
    - Con memoria distribuida (NUMA).- Menor latencia – escalable pero requiere para ello distribución de datos/código
- Criterio de clasificación: nivel de empaquet./conexión
    - Sistema
    - Armario(cabinet)
    - Placa(board)
    - Chip

· Multiprocesador en una placa: evolución de UMA a NUMA (Pg.- 10.11.12)

- UMA
    - Controlador de memoria en chipset (Northbrigde chip)
    - Red: bus (medio compartido, la usan las CPU para comunicarse)
- NUMA
    - Controlador de memoria en chip del procesador (en cada CPU)
    - Red: enlaces (conexiones punto a punto) y conmutadores (en el chip del procesador)

|Multicores|

- Pueden tener varias estructuras:
    - L1 y L2 independientes, L3 compartida, con conexión a un conmutador que permite el acceso a el controlador de memoria o controladores de memoria en caso de tener mas de uno.
    - Caches independientes, conectadas por bloques a la LastLevelCache, conectaca a su vez al conmutador.
    - Caches independientes, inclusive la LastLevelCache, todas se conectan al conmutador y este a memoria.
    - Caches indpendientes, se conectan a un conmutador que a su vez se conecta a memoria principal y a una LastLevelCache

|Cores Multithread|

· Arquitecturas ILP

- Escalar segmentada.-Procesadores/Cores segmentados.- Ejecutan instrucciones concurrentemente segmentando el uso de sus componentes
- VLIW y superescalar.- Procesadores/cores VLIW (Very Large Instruction Word) y superescalares.- Ejecutan instrucciones concurrentemente (segmentación) y en paralelo (tienen múltiples unidades funcionales y emiten múltiples instrucciones en paralelo a unidades funcionales)
    - VLIW:
        - Las instrucciones que se ejecutan en paralelo se captan juntas de memoria.
        - Este conjunto de instrucciones conforman la palabra de instrucción muy larga a la que hace referencia la denominación VLIW
        - El hardware presupone que las instrucciones de una palabra son independientes: no tiene que encontrar instrucciones que pueden emitirse y ejecutarse en paralelo.
    - Superescalares:
        - Tiene que encontrar instrucciones que puedan emitirse y ejecutarse en paralelo (tiene hardware para extraer paralelismo a nivel de instrucción)

· Modificación de la arquitectura ILP en Core Multithread (ej. SMT) pg.-20

- Almacenamiento: se multiplexa, se reparte o comparte entre threads, o se replica
    - Con SMT: repartir, compartir o replicar
- Hardware dentro de etapas: se multiplexa, o se reparte o comparte entre threads
    - Con SMT: unidades funcionales (etapa Ex) compartidas, resto etapas repartidas o compartidas; multiplexación es posible (p. ej. predicción de saltos y decodificación)

· Clasificación de cores multithread

- Temporal Multithreading (TMT).-
    - Ejecutan varios threads concurrentemente en el mismo core
    - La conmutación entre threads la decide y controla el hardware
    - Emite instrucciones de un único thread en un ciclo
- Simultaneous MultiThreading (SMT) o multihilo simultáneo o horizontal multithread
    - Ejecutan, en un core superescalar, varios threads en paralelo
    - Pueden emitir (para su ejecución) instrucciones de varios threads en un ciclo

· Clasificación de cores con TMT

- Fine-grain multithreading (FGMT) o interleaved multithreading.- La conmutación entre threads la decide el hardware cada ciclo (coste 0). (Rotatorio, round robin, 1 hebra en cada ciclo de reloj)[1][2][3][1][2][3]
- Coarse-grain multithreading (CGMT) o blocked multithreading.- La conmutación entre threads la decide el hardware (coste de 0 a varios ciclos). (Intervalos, slice o eventos)[1][1][1][ ][2][2][2][3][3][3]

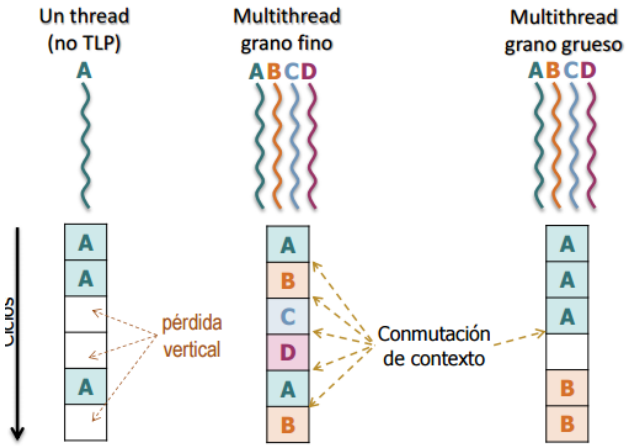
· Clasificación de cores con CGMT con conmutación por eventos

Estática:

- Conmutación
  - Explícita: instrucciones explícitas para conmutación (instrucciones añadidas al repertorio)
  - Implícita: instrucciones de carga, almacenamiento, salto
- Ventaja/Inconveniente:
  - coste cambio contexto bajo (0 o 1 ciclo) / cambios de contextos innecesarios

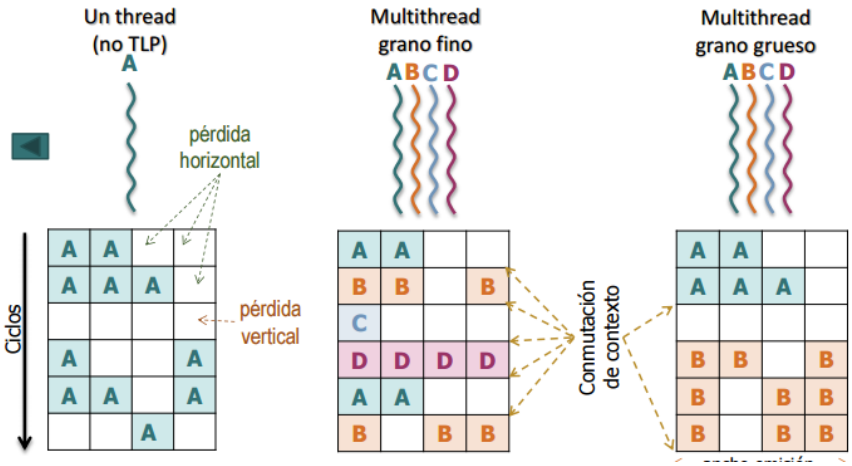
Dinámica:

- Conmutación típicamente por:
  - fallo en la última cache dentro del chip de procesamiento (conmutación por fallo de cache), interrupción (conmutación por señal), …
- Ventaja/Inconveniente:
  - reduce cambios de contexto innecesarios / mayor sobrecarga al cambiar de contexto

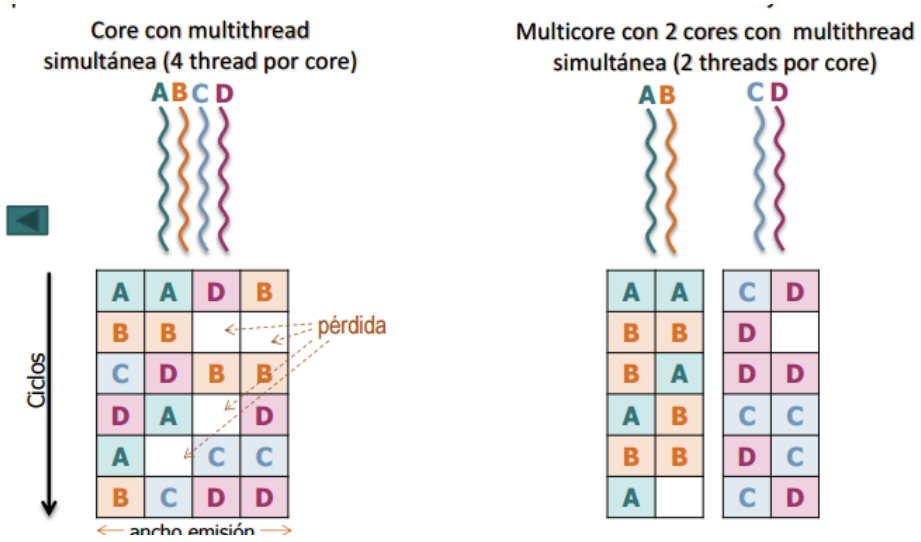


En un core escalar se emite una instrucción cada ciclo de reloj

En un core superescalar o VLIW se emiten más de una instrucción cada ciclo de reloj; en el dibujo, de un único thread



En un multicore y en un core superescalar con SMT (Simultaneous MultiThread) se pueden emitir instrucciones de distintos threads cada ciclo de reloj



|Hardware y arquitecturas TLP en un chip|

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

Lección 8. Coherencia del sistema de memoria

|Computadores que implementan hardware mantenimiento de coherencia|

CC-NUMA(NUMA) COMA(NUMA) SMP(UMA)

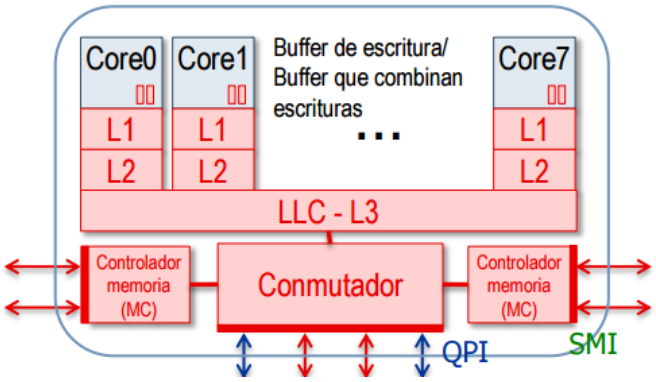
|Sistema de memoria en multiprocesadores|

¿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)



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

·Incoherencia en el sistema de memoria se puede dar por varias situaciones, como:

Clases de estructuras de datos	Causa de problemas por falta de coherencia	Falta de coherencia
Datos modificables	E/S	Cache-MP
Datos modificables compartidos	Fallo de cache	Cache-MP
Datos modificables privados	Emigra proceso→Fallo cache	Cache-MP
Datos modificables compartidos	Lectura de cache no actualizada	Cache-Cache

· 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

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

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

Aun con estos métodos, se pueden producir situaciones de incoherencia, si por ejemplo dos procesadores envían escritura con invalidación en dos instantes antes de comunicarse entre ellos puede dar lugar a que cada uno tenga un dato.

· 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

Básicamente es un Vector de bits con información sobre cachés con copia así como el estado de los bloques en memoria

· 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

MIRAR PAGINAS 18-23 Leccion 8

| Protocolos de mantenimiento de coherencia: clasificación y diseño |

· 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

|Protocolo MSI de espionaje|

- 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

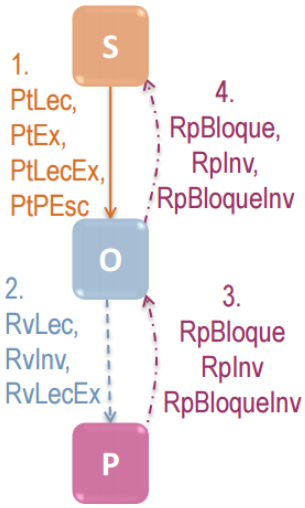
EST. ACT.	EVENTO	ACCIÓN	SIGUIENTE
Modificado (M)	PrLec/PrEsc		Modificado
	PtLec	Genera paquete respuesta (RpBloque)	Compartido
	PtLecEx	Genera paquete respuesta (RpBloque) Invalida copia local	Inválido
	Reemplazo	*Genera paquete posescritura (PtPEsc)*	*Inválido*
Compart. (S)	PrLec		Compartido
	PrEsc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

|Protocolo MESI de espionaje|

- 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

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	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

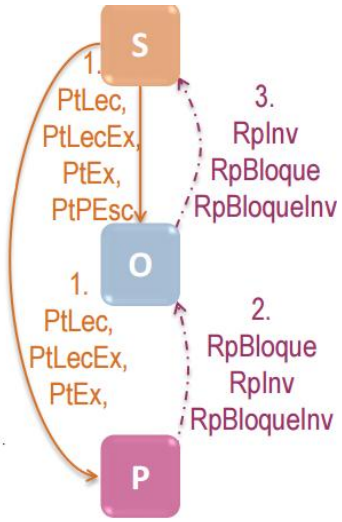
|Protocolo MSI basado en directorios con o sin difusión|



· MSI con directorios (sin difusión)

- 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)

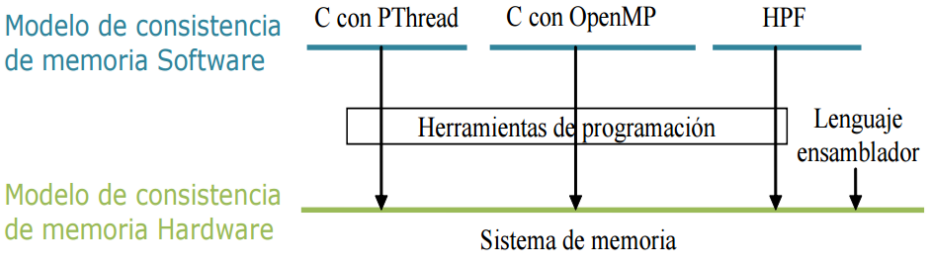
· MSI con directorios (con difusión)



- 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 (RpInv)
  - nodo O a S: resp. con bloque (RpBloque), resp. con o sin bloque confirmando fin inv. (RpInv, RpBloqueInv)

Lección 9. Consistencia del sistema de memoria

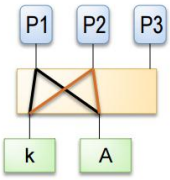
| Concepto de consistencia de memoria |



- 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

| 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 globa



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

| 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→R, W→W,R→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.

|Modelo que relaja W->R|

- Permiten que una lectura pueda adelantar a una escritura previa en el orden del programa; pero evita dependencias RAW
  - Lo implementan los sistemas con buffer de escritura para los procesadores (el buffer evita que las escrituras retarden la ejecución del código bloqueando lecturas posteriores)
  - Generalmente permiten que el procesador pueda leer una dirección directamente del buffer (leer antes que otros procesadores una escritura propia)
- Para garantizar un orden correcto se pueden utilizar instrucciones de serialización
- Hay sistemas en los que se permite que un procesador pueda leer la escritura de otro antes que el resto de procesadores (acceso no atómico)
  - Para garantizar acceso atómico se puede utilizar instrucciones de lectura-modificación-escritura atómicas

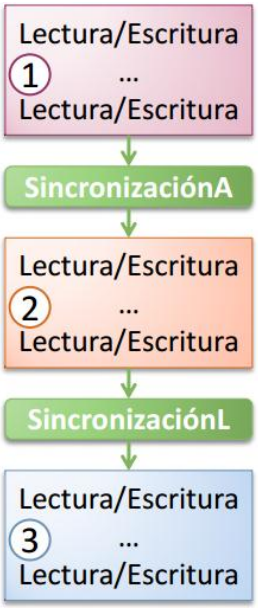
| Modelo que relaja W->R y W->W |

- Tiene buffer de escritura que permite que lecturas adelanten a escrituras en el buffer
- Permiten que el hardware solape escrituras a memoria a distintas direcciones, de forma que pueden llegar a la memoria principal o a caches de todos procesadores fuera del orden del programa.
- En sistemas con este modelo se proporciona hardware para garantizar los dos órdenes. Los sistemas con Sparc implementa un modelo de este tipo.
- Este modelo no se comporta como SC en el siguiente ejemplo:

P1	**P2**
A=1; k=1;	while (k=0) {}; copia=A;

| Modelo de ordenación débil |

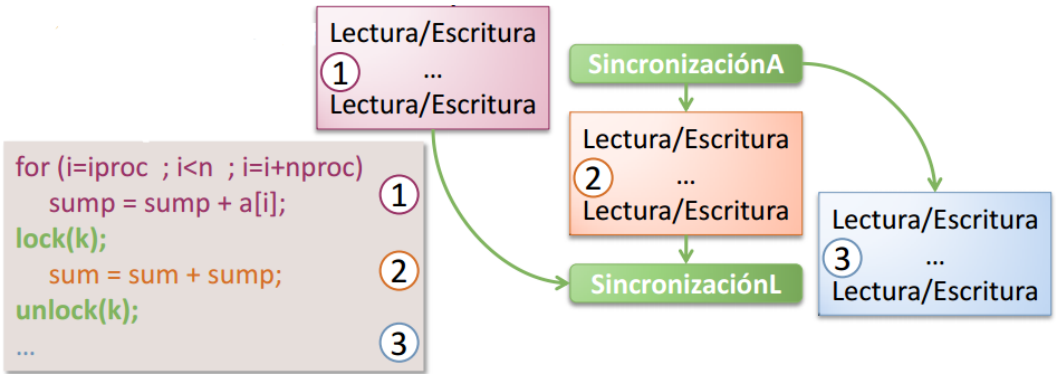
- Relaja W->R, W->W y R->RW
  - Si S es una operación de sincronización (liberación o adquisición), ofrece hardware para garantizar el orden:
    - S->WR
    - WR->S
- PowerPC implementa un modelo basado en ordenación débil




```
for (i=iproc  ; i<n  ; i=i+nproc)          1
    sump = sump + a[i];
lock(k);
    sum = sum + sump;      /* SC */        2
unlock(k);
...                                        3
```

| Consistencia de liberación |

- Relaja W->R, W->W y R->RW
  - Si SA es una operación de adquisición y SL de liberación, ofrece hardware para garantizar el orden:
  - SA->WR y WR->SL
- Sistemas con Itanium implementan un modelo de consistencia de liberación



Lección 10. Sincronización

| Comunicación en multiprocesadores y necesidad de usar código 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**

...
A=1;
K=1;
...
	...
	while (K=0) { };
	copia=A;
	...

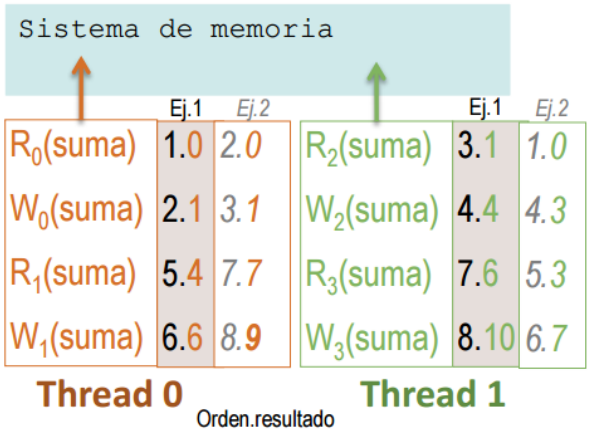
· Comunicación colectiva

Secuencial	**Paralela** (sum=0)

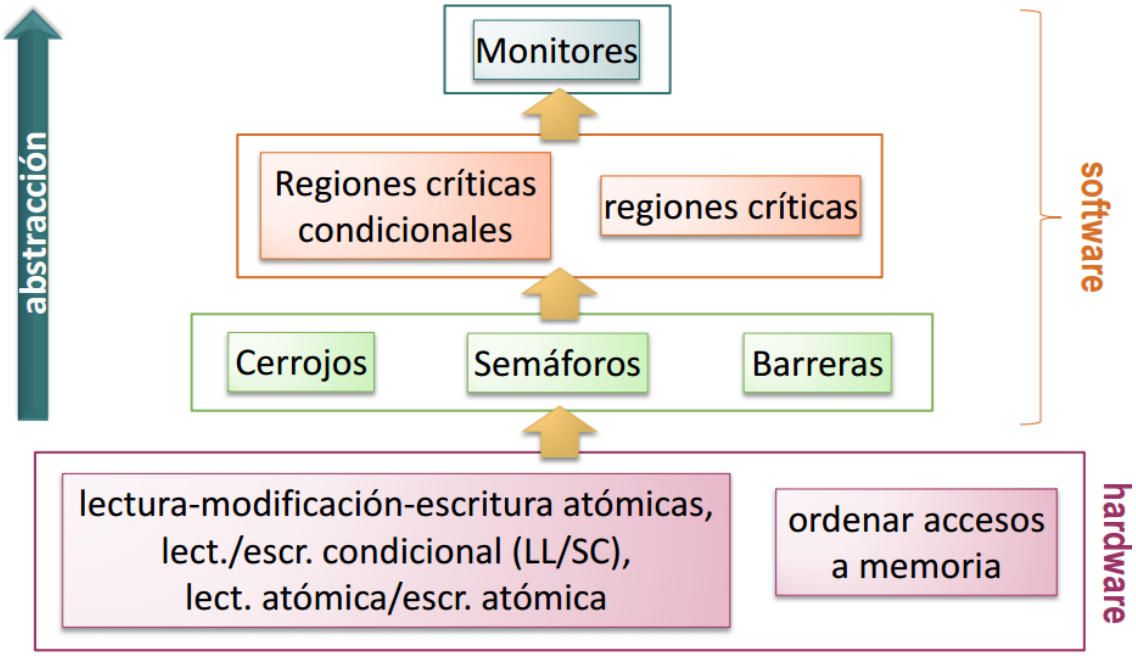
for (i=0 ; i<n ; i++) {
sum = sum + a[i];
}
	for (*i=ithread* ; *i*<n ; *i=i*+nthread) {
	sump = *sump* + a[*i*];
	}
	sum = **sum** + *sump*; /* SC, sum compart. */
	if (*ithread*==0) printf(**sum**);

- 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)



| Soporte software y hardware para sincronización |



| 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.

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);
	sum = **sum** + *sump*; /* SC, sum compart. */
	unlock(k);

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

· 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;
```

|Barreras (con problema reutilización)|

```
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 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
}
```

| Apoyo hardware a primitivas software |

· Instrucciones de lectura-modificación-escritura atómicas (cerrojos)

Todas las instrucciones son atómicas y se pueden usar como cerrojos por esto mismo.

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)

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 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]
          cmpwi   r5,0   #si está cerrado (a 1)
          bne-    bucle  #esperar en el bucle, en caso contrario
          stwcx.  r4,0,r3  #poner a 1 (r4=1): M[r3] ← r4
          bne-    bucle  #el thread repite si ha perdido la reserva
          isync          #accede a datos compartidos cuando sale del bucle
```

```
unlock:                  # unlock(M[r3])
         sync            #espera hasta que terminen los accesos anteriores
         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 */
```