GuiaSupervivenciaAC-Tema3.pdf

Erwapo

Arquitectura de Computadores

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Guia de supervivencia de Arquitectura de ordenadores.
Tema 3: Arquiteturas paralelas a nivel de hebra.

Lección 7: Arquiteturas TLP

-7.1 Clasificación de arquitecturas con TLP explícito y una instancia del SO.

Antes de adentrarnos en la clasificación de estas arquitecturas de arquitecturas TLP, hay que recordar que al referirnos al tipo explícito, hacemos referencia que los flujos de instrucciones son creados y gestionados por el sistema operativo, y al ser de una sola instancia de SO, nunca hablaremos de multicomputadores, solamente de multiprocesadores, multicores y de cores multithread.

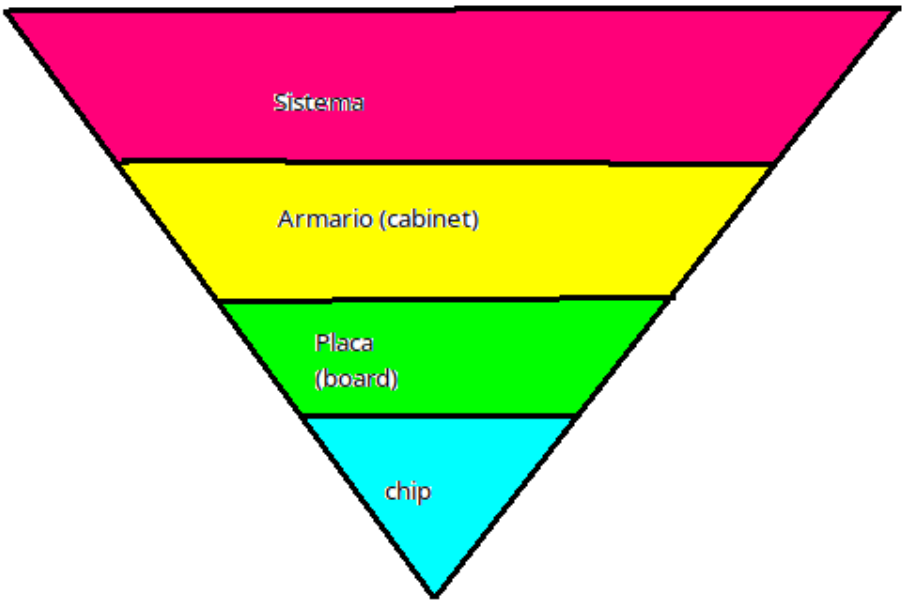
- Multiprocesadores: llamaremos así a los computadores con varios procesadores/núcleos que permiten varios flujos de instrucciones en paralelo(es decir, cada hebra se encontrará en cada núcleo o procesador disponible). Cabe resaltar que un multiprocesador se puede encontrar en diferentes tipos de empaquetamiento: podemos encontrar un un multiprocesador desde en un dado de silicio hasta en un sistema informático completo.

-Multicore o multiprocesador en un chip: se trata de un chip con múltiples núcleos, que permiten ejecutar paralela varios flujos de instrucciones (tendremos al menos un flujo de instrucción por cada núcleo).

-Core multithread: es un núcleo de procesamiento que debido a su arquitectura a nivel de instrucción, permite varios flujos de instrucciones a la vez, ya sea porque utilice segmentación de cauce, un sistema superescalar o un VLIW (que se tratarán en el tema 4).

-7.2 Multiprocesadores
A la hora de clasificar los multiprocesadores, los haremos en 2 tipos de clasificación: nivel de empaquetamiento, y según su sistema de memoria.

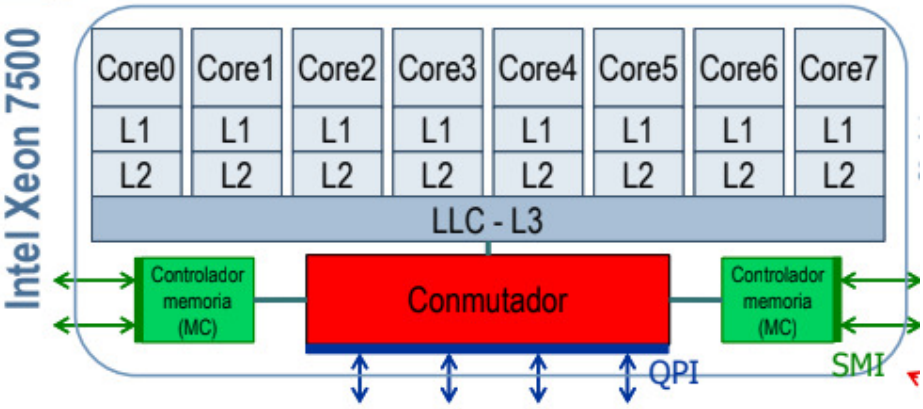
A nivel de empaquetamiento: como se ha mencionado anteriormente en el 7.1, los multiprocesadores cuentan con gran cantidad de niveles de tamaño/empaquetamiento, presentándose en lo siguientes niveles: Sistema, armarios (o cabinets), placa o incluso se pueden presentar comoun solo chip de procesamiento:



A nivel de sistema de memoria podemos encontrar 2 grandes grupos de los que tratamos en el tema 1: los UMA, que cuentan con la memoria repartida de forma equitativa a todos los procesadores, por lo que cada procesador tardará lo mismo en acceder a cualquier espacio de memoria, este sistema de memoria presenta muy poca escalabilidad, pues añadir más procesadores no aporta una mejor velocidad a la hora de trabajar con algún trozo de memoria, también nos lo podemos encontrar presentados como SMP (symmetric multiprocessor); también podemos encontrarnos con multiprocesadores tipo NUMA, que cuentan con la memoria repartida de tal forma el acceso a esta varia según la localización de los procesadores, por lo que el colocar un nuevo procesador aumenta en mayor medida las prestaciones del sistema, pues se podrá acceder de forma más rápida al rango de memoria que controle ese procesador, el incoveniente de este tipo de procesadores se encuentra en la distribuciónde datos y código, pues dificulta el poder programar en este tipo de computadores sacando el máximo rendimiento posible, entre los tipo de NUMA tenemos los NUMA, los CC-NUMA y los COMA.

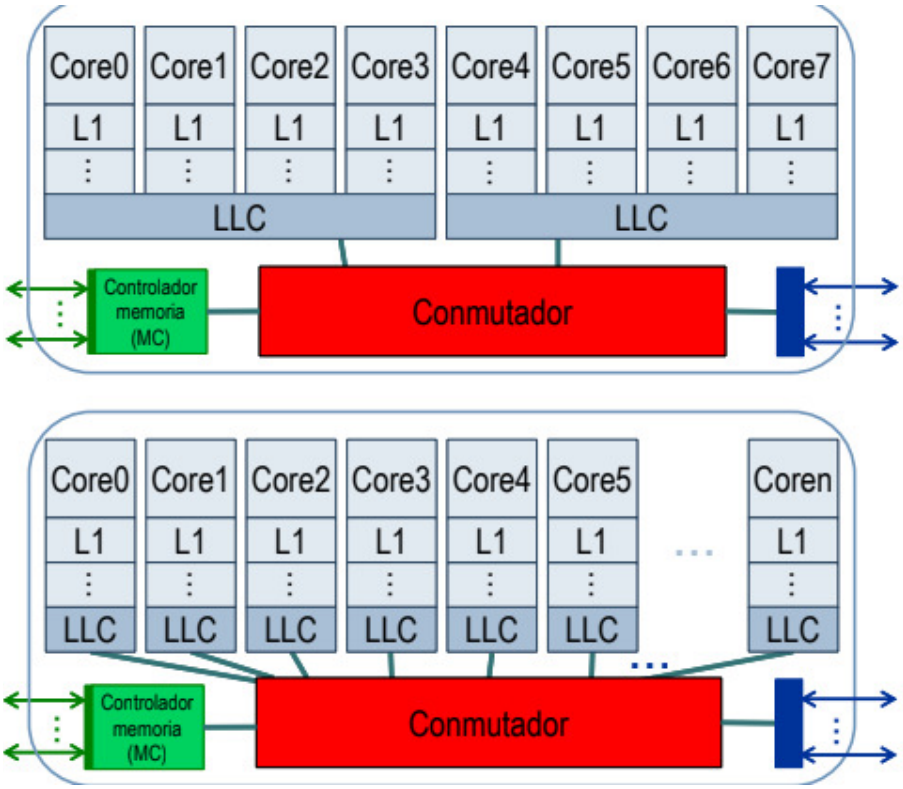
-7.3 Multicores

Sobre los chip multinúcleo tenemos destacar la ordenación de las cachés, así pues algunas arquitecturas cuentan con un solo bloque de caché L3 que después se reparte en otras líneas de caché de nivel L2 y L1, así pues podemos encontrar algunas arquitecturas que presentan formas así:



o así:

(no hay demasiada chicha sobre este bloque la verdad)



-7.4 Cores multithread

Como hemos comentado antes, los cores multithread son núcleos de procesamiento que debido a su arquitectura, permiten varios flujos de instrucción, ya sea de forma concurrente, o paralela. Así pues podemos encontrar 3 tipos de arquitecturas que nos llevan a la ejecución de varias hebras simultáneas:

-Sistemas segmentados: para que una instrucción se lleve a cabo, hace falta realizar una serie de pasos, en los que podemos subdividir la ejecución del programa, gracias a eso podemos tener a una instrucción ejecutándose por étapa.

- Sistemas superescalares: no todas las instrucciones utilizan los mismo recursos del ordenador, mientras que algunos necesitan de una ALU para poder realizar una operación, otras puede requerir solo de un acceso a memoria, así pues un sistema superescalar permite ejecutar dos instrucciones que necesiten de diferentes recursos del ordenador (en caso de utilizar el mismo recurso, se produciría un riesgo estructural, que provocaría un caso de cuello de botella, significando en un retardo en la ejecución del programa). Cabe destacar que tanto los sistemas superescalares como los VLIW pueden actuar de forma segmentada, consiguiendo aún más ganancia de prestaciones.

- Sistema VLIW (very long instruction word): este sistema lo que hace es agrupar varias instrucciones diferentes y juntarlos como si fuesen una sola instrucción, permitiendo que se realicen todas esas instrucciones en el tiempo (general) en el que actúa una sola instrucción. Se suelen encontrar más en sistemas empotrados.

-7.4.1 Modificación de la arquitectura ILP en cores multithread

Podemos clasificar los ńucleos multithread en los siguientes tipos:
-Temporal multithreading (TMT): es unos de los métodos principales de multithreading que consisten en ejecutar los threads de manera **concurrente**, por lo que durante un momento dado, el núcleo solo emitirá de instrucciones de una sola hebra hasta que el hardware decida conmutar las threads. Tmabién se les puede ver llamados como super-threading.

- Simultaneous multithreading (SMT): en este caso las hebras sí se ejecutan de forma paralela, por lo que en un ciclo se puede ejecutar instrucciones de 2 hebras diferentes en un ciclo (en los TMT son varias instrucciones de un mismo thread en un mismo ciclo), y no requiere de un sistema de conmutación como el método TMT.

Dentro del sistema multithreading tenemos que preguntarnos una cosa: si en cada momento solo se está ejecutando una sola hebra y el hardware decide cuando cambiar de hebra ¿ cada cuánto se produce la conmutación? Pues bien, tenemos 2 tipos de metodología TMT:

- De grano fino(FGMT): En este tipo de caso la conmutación de las hebras se produce en cada ciclo de reloj del computador, haciendo que tras cada ciclo de reloj se cambie de hebra en ejecución, como estas conmutaciones van a ser muy comunes, es de intuir que tenemos que usar metodologías de manejo rápido para poder escoger cuál es la siguiente hebra a ejecutar, por lo que se suelen escoger con metodologías tipo round-robin o escoger la hebra que menos ciclos necesite para finiquitar.

- De grano grueso(CGMT): En este tipo de caso la conmutación se llevará a cabos tras varios ciclos de instrucción, estas conmutaciones se llegarán a cabo ya sea por que se ha sobrepasado un límite de tiempo prefijado (timeslice multithreading) o por que se lleva a cabo un evento específico que involucra un cambio de hebra (swicth-on-event multithreading). Este tipo de eventos los podemos clasificar en 2 grupos principales: estáticos y dinámicos.

Los eventos estáticos se generan porque el programa señala de alguna manera, que hace falta dejar de utilizar esta hebra, por lo que encontramos 2 tipos de eventos estáticos, los explícitos que son instrucciones del repertorio que señalan únicamente que tenemos que realizar una conmutación, y los explícito, suelen ser instrucciones de salto, carga de memoria o almacenamiento, que aunque no necesiten de una conmutación para llevarse a cabo, es mejor para el hardware realizar un cambio de hebra en este tipo de casos, por lo que viene implícito en la instrucción. Este tipo de instrucciones suelen tardar de uno a 2 ciclos en realizar el cambio (que para los CGMT es poco) aunque suele provocar muchos cambios de hebra que en algunos casos, no es necesario.

También están los eventos dinámicos que se generan por algún evento relacionado por el hardware, por ejemplo si se ejecuta un fallo de caché y hace falta cargar memoria en caché, conviene que otra hebra cambie a ejecutarse mientras cargamos esa nueva información, este tipo de eventos necesita de una mayor sobrecarga para realizar cambios de contexto, por lo que suele tardar varios ciclos de reloj en ejecutarse, pero estos cambios de hebras sí tenderán a ser muy necesarios.

Lección 8 coherencia del sistema de memoria

Pequeña anotación antes de empezar con la lección: a la hora de hablar de mutliprocesadores que mantienen coherencia a través del hardware, hablamos de los CC-NUMA, los COMA y los SMP, tanto los multicomputadores como los NUMA normales no cuentan con esta adición.

8.1 Sistema de memoria en multiprocesadores

A la hora de hablar de la consistencia en memoria tenemos que tener en cuenta que conocemos como sistema de memoria, este se forma por una red de interconexión que denominaremos bus, que conectarán tanto los procesadores, que cada uno contendrá varios niveles de caché con los que trabaja, la memoria principal que contendrá toda la información del computador, y los bufferes tanto de escritura, como de lectura.

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

Para poder aprovechar la máxima capacidad de un computador hay que utilizar cada uno de los procesadores que tenemos para obtener el máximo rendimiento, a la hora de trabajar con información cada procesador cuenta con su propia caché que irán rellenando de manera temporal con la información que encuentren en memoria principal, toda información que la memoria principal provee a las caches actúan como copias de la que se encuentran en memoria principal, así que puede suceder que haya procesadores con un mismo espacio de memoria pero que ambos hayan registrado diferentes valores a esa variable, por lo que es muy importante que la **coherencia** de la infomración que se encuentra en los procesadores sea uniforme y que si un nuevo procesador carga en memoria esa nueva información, está represente la que contienen el resto de procesadores. Así pues podemos encontrar 4 tipos de incoherencias:

Clases de estructuras de datos	**Eventos que ponen de manifesto la incoherencia**	**Tipo de Falta de coherencia**
Datos modificables	E/S	Caché-memoria
Datos modificables compartidos	Fallo de caché	Caché-memoria
Datos modificable privados	Emigra thread/proceso → Fallo de caché	Caché-memoria
Datos modificables compartidos	Lectura de caché no actualizada	Caché-caché

Smilke®

PARA ESTE EXAMEN AÚN TE FALTAN UN PAR DE CAFÉS MÁS. DISFRÚTALOS COMO SE MERECEN.

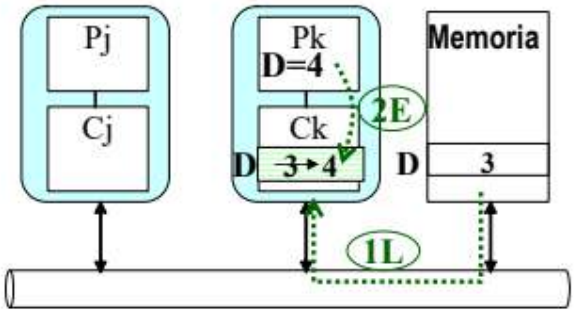
Descuentazo para tí

CÓDIGO: WUOLAH10

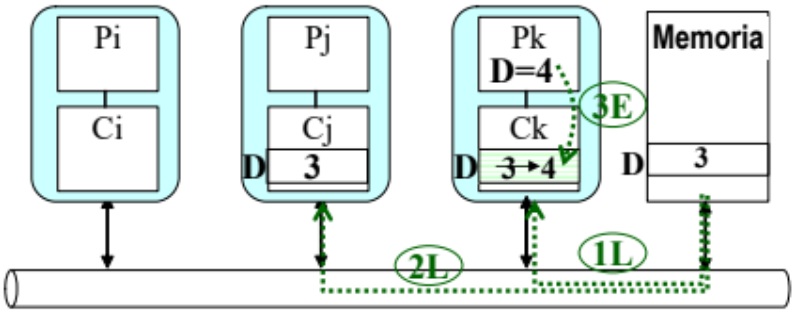
COMPRA AQUÍ

Procederé a poner 2 ejemplos de incoherencia para que podáis entender como se dan este tipo de fallos:

Un caso en el qu tenemos 2 procesadores: Pj y Pk, Pk quiere leer el valor de D, una variable que no tiene en caché. Así que solicita a la memoria principal obtener una copia de esta, una vez obtenido esta copia de D, que vale 3 tanto en la caché de Pk como en la memoria, el procesador Pk procede a cambiar su valor, esto genera un incoherencia pues los valores de D que se encuentra en Pk y memoria principal no son iguales.



Otro caso por ejemplo es tener 2 procesadores: Pj y Pk, ambos quieren leer el valor D, así que ambos acceden a la memoria principal para obtener el valor de este valor. D en ese momento vale 3, así que los valores de D en ambas cachés valdrán 3, una vez obtenido estas variables, el procesador Pk decide modificar D, dándole el valor de 4; vemos en este caso que estamos ante un estado de incoherencia pues el valor que hay tanto en Pj como en la memoria principal (que significa que si otro procesador necesita del valor D, le dará el valor desactualizado) no es válido.



8.2.1 Métodos de actualización de memoria principal implementados en cachés

Como hemos visto, tenemos que evitar a toda costa la existencia de incoherencia en nuestro computador, para evitar esto necesitamos en primer lugar como hacemos para actualizar la memoria principal con los cambios de las variables en caché, así pues encontramos 2 métodos:

-Escritura inmediata: tan simple como hacer que cada vez que un procesador sobrescriba en caché algún valor, también guardaremos su nuevo valor en memoria principal.

-Posescritura: lo que sucede con la escritura inmediata es que es muy poco eficiente pues tenías al bus constantemente mandando información sobre actualizaciones de variables, debido a este problema se ideo la posescritura, al cargar en caché un bloque de memoria, por principios de localidad temporal y espacial sabemos son posiblemente los próximos en ser modificados, así pues la posescritura solo pedirá actualizar la memoria principal cuando todo el bloque de caché es desalojado.

8.2.2 Alternativas para propagar una escitura en protocolos de coherencia de caché

Una vez obtenido un método para actualizar la memoria principal, queda idear métodos para actualizar la información modificada en las cachés que cuenten con la bloque de memoria modificado:

- Escritura con actualización: es parecido a la metodología de escritura inmediata, cada vez que se escribe un valor en un procesador, este comunica a los demás procesadores que cuenten con una copia de ese dato para que actualicen su valor también. Como sucedía con la escritura inmediata, esto satura los buses de comunicación, por lo que se creo la escritura con invalidación.

-Escritura con invalidación: en este caso lo que haremos será que al momento de que se realice una escritura en caché, mandaremos un mensaje para invalidar las copias que hayan en las demás caché, reduciendo así el tráfico en el bus.

Ahora bien, dado ya estás alternativas para asegurar la actualización de datos en caché, sigue pudiendo presentarse incoherencias en los sistemas: por ejemplo supongamos que contamos con un NUMA con 4 procesadores: P0,P1,P2 y P3, de tal forma que P0 escribe en el valor A, que se encuentra en todos los procesadores, y le da valor 2, pero a su vez el P1 escribe su valor 2, al tratarse un NUMA puede suceder que el aunque el orden de instrucciones sea primero A=1 y luego A=2, suceda que a estar a diferentes distancias unos procesadores de otros, ocurra que a algunos procesadores le llegue primero la orden de A=2 y luego A=1, llevando a cabo una situación de incoherencia.

Para evitar que un sistema pueda evitar incoherencias necesitaremos cumplir los requisitos explicados más adelante.

8.2.3 Requisitos del sistema de memoria para evitar problemas de incoherencia

Tendremos 2 requisitos fundamentales a la hora de evitar coherencias, el **propagar** de manera correcta para lleguen a los procesadores que cuentan con una copia de ese

valor, y **serializar** las escrituras de tal forma que correspondan con el orden en el que se generaron.

Aplicación de propagación y serialización en un bus:

- En este caso tanto propagación y serialización no cuentan con problemas algunos a la hora de usar los métodos de actualización de la 82.2 y la 8.2.3, los buses conectan a todos los procesadores y memorias que haya, así que a la hora de propagación toda actualización es visible para todos los nodos conectados (los controladores de caché serán los encargados de ver si la actualización en el bus corresponde a un bloque que contiene la caché del nodo), respecto a la serialización es más simple aún que la propagación, pues al tratarse de un solo bus para todos los nodos, las actualizaciones siempre se mostrarán en el orden mostrado (en el bus solo habrá una actualización).

Aplicación de propagación y serialización en redes que no sean buses:

- A la hora de la propagación podremos usar 2 métodos, difusión, que consiste en que los paquetes de actualización/invalidación se mandan a cada una de las cachés disponibles, o bien, si queremos enviar el paquete solamente a las cachés que contengan contenga una copia del bloque de memoria (lo que aumenta la escabilidad del multiprocesador), crearemos un directorio para cada uno de los bloques, en el que se mostrarán los nodos que contengan una copia de ese bloque, este directorio puede ser centralizado, lo que implica que es compartido por todos los nodos y que contenga la información de todos los bloque s de memoria, o bien distribuido, distribuido, en el cada nodo contará con su propio directorio, en el que solo mostrará la información de los nodos que tengan cargados en sus memorias,

- En este tipo de casos la serialización se consigue a través de una actualización supervisada, lo que haremos será que uno de los nodos, al que denominaremos home, será el encargado de controlar y serializar las actualizaciones que se realizan en el sistema.

-8.3 Protocolos de mantenimiento de coherencia

Una vez estudiado todo lo relativo a los requisitos y métodos que disponemos para generar un sistema de memoria sólido libre de incoherencias, es momento de ver definir protocolos, los protocolos es la manera de actuar de la manera de actuar de todo el sistema de memoria para mantener la coherencia de manera constante.

8.3.1 Clasificación de protocolos

-Protocolos de espionaje (snoopy): en estos sistemas se propaga a través de difusión y es el hardware asociado a las cachés los que se encargan de espiar la información del bus conectado, es ideal para multiprocesadores basados en buses o bien la difusión

es muy eficiente (ya sea porque el número de nodos es pequeño o por que la red permite una difusión muy veloz).

- Protocolos basados en directorios: estos protocolos están hechos para minimizar el tráfico en las redes de conexión, ya solo se envía mensaje a los nodos que cuenten con la copia de memoria con la que se este trabajando, para realizar esto se utilizará tanto la propagación como la serialización para redes que no son buses explicado en el 8.2.2, este sistema es ideal para procesadores tipo CC-NUMA o COMA (recordad que los NUMA normales no cuentan con sistema de coherencia) y con UMA de red escalable (dance-hall).

-Protocolo jerarquicos: están hecho para computadores que muestren una jerarquia de buses o redes escalares. (No son demasiados importantes, no hace falta saber más).

8.3.2 Facetas de diseño lógico en protocolos de coherencias

Este apartado es bastante simple, vamos a observar que cosas necesitamos para diseñar un protocolo:

-1 política de actualización de memoria principal: podremos usar el método de escritura inmediata, posescritura o bien realizar una versión mixta. (Generalmente se prefiere la posescritura).

- 2 política de coherencia entre cachés: tendremos la escritura con invalidación, escritura con actualización o otra versión mixta, (Se suele preferir la escritura con invalidación).

-3 describir el comportamiento: este es el punto más importante a la hora de generar un sistema lógico, primero de todo tendremos que definir una serie de estados en los que clasificaremos los bloques de memoria, así podremos tener información y control sobre cada uno de los bloques que tengamos, una vez definido los estados toca ver que tipo de transferencias va a realizar nuestro sistema y catalogarlo, esto será fundamental pues una vez terminado podremos realizar el paso final, que será definir las transiciones de los estados a otros, por lo que un tipo de mensaje concreto podrá cambiar un estado a otro dependiendo del estado final, esto nos permitirá organizar el sistema de coherencia de una forma correcta.

- 8.4 Protocolo de espionaje de 3 estados (MSI)

El MSI es un tipo de protocolo de espionaje, por lo que es importante recordar que trabajaremos con difusión, tendremos 3 tipos de estado para un bloque en caché:

- Modificado(M): este estado significa que es la única copia válida del sistema, esto sucederá cuando se cargue un valor en caché y este dato en memoria vayamos a modificarlo en esa caché, a partir de la modificación tanto las copias de las demás

cachés como de la memoria principal son incorrectos, así que se deberá cambiar el resto de estados y dejar este último nodo como modificado, y en caso de que otro nodo requiera de esa información, será el nodo modificado y no la memoria principal la que encargue de darle el valor correcto al nodo solicitante.

- Compartido(S): este estado se utilizará para señalar de que el bloque de información es correcto pero no es el único sitio donde es válido este valor también (ya sea porque la memoria principal cuenta con el valor correcta o otra caché la contiene), este estado es muy común para casos de lectura pero no escritura.

- Inválido: el valor del bloque no se encuentra en la caché del nodo o se ha invalidado.

También contamos con 2 estados para la memoria principal:
-Válido: el valor que se encuentra en la memoria principal está actualizada y tiene el valor correcto.
- Inválido: el valor no está actualizado, probablemente se deba por una caché a modificado el valor del bloque, por lo que la versión válida se encuentra en la caché en estado Modificado.

En lo referentes a las transferencias de información del protocolo encontramos las siguientes:

Referente a lectura tendremos 2 transferencias: las peticiones de lecturas normales, que usaremos cuando encontremos un fallo de caché y queramos obtener el bloque a buscar, y la petición de acceso exclusivo, al que llamaremos cuando tengamos un fallo de caché sobre un bloque de memoria que queramos modificar, en este caso habrá habrá que invalidar el resto de copias de ese bloque de memoria.
Referente a la escrituras tenemos la petición de posescritura, esta la invocaremos cuando vayamos a reemplazar el bloque de memoria, esto es importante si tenemos la caché con bloque en estado modificado, pues la versión que se encuentra en memoria principal no es correcta pero se va a reemplazar por una correcta.

Finalmente quedaría la transferencia de un bloque solicitado por una caché, está puede llegar desde la propia memoria principal o de una caché que cuente con una copia válida del bloque.

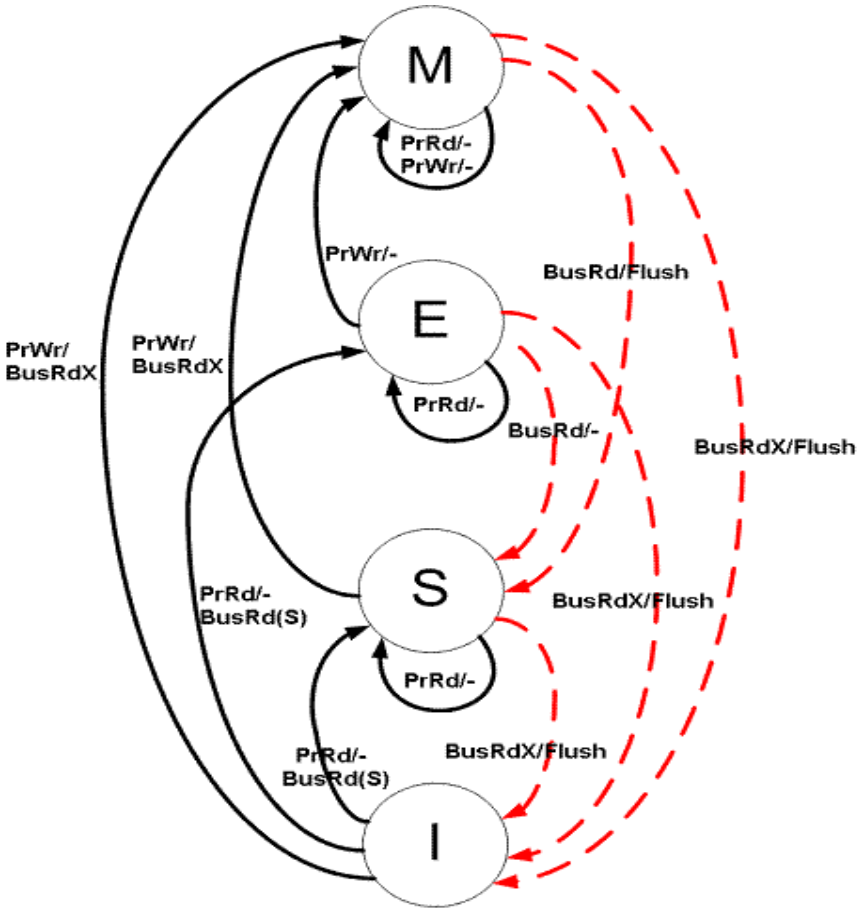
A modo de resumen voy a dejar dejar una foto con la tabla de eventos posibles según el estado de un bloque:

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

- 8.5Protocolo de espionaje de cuatro estados (MESI)

Este protocolo es IDÉNTICO al protocolo MSI salvo por la excepción de que contará con un estado extra, el estado exclusivo. Va a mantener la política de posescritura y sistema de escritura con invalidación.

Como hemos visto anteriormente, cuando se escribía una variable en una caché es necesario mandarle una señal de invalidación al resto de nodos para que en caso de contar con una copia del bloque de memoria, lo invalidase, pero esto lastra mucho a los programas secuenciales ejecutadas en un multiprocesador, ya que en estos casos no se suele compartir variables entre procesos, para evitar esta degradación de prestaciones se añade un nuevo estado: el exclusivo, un bloque estará en este estado única y exclusivamente cuando solamente hay un nodo en con este bloque de memoria en caché (como el estado Modificado) pero también el bloque es correcto en la memoria principal. Esto ocurrirá cuando en un nodo ocurra un fallo de caché y necesite el bloque de memoria en caché, este realice una petición de lectura a la memoria principal, y el controlador de la memoria vea que no hay ningún otro nodo que tenga o requiera de esta memoria. Las ventajas que presenta esto es que si un nodo tiene un bloque en estado exclusivo y quiere escribir sobre él, no hace falta mandar un mensaje de invalidación a ningún otro nodo del sistema, pues sabemos a ciencia cierta que el nodo solicitante es el único con el trozo de memoria cargado.



-8.6.1 MSI basado en directorios sin difusión

Este sistema mantiene los estado del protocolo MSI, pero ahora no nos encontramos en un sistema de un solo bus (o el número de nodos es muy alto), por lo que el sistema de propagación no será difusión, sino que usaremos propagación basado en directorios, a su vez, utilizaremos para la serialización el uso del nodo controlador, por lo que contaremos con un nodo home que actuará como controlador de las llegadas de las peticiones.

Respecto a los estados, como he comentado antes, siguen idénticos, tenemos el estado de modificado, compartido e inválido para memoria en caché y válido e inválido para memoria principal.
Las transferencias si sufren cambios, pues al no tener espionaje, hay que invalidar los directorios a través de peticiones así pues tendremos las siguientes transferencias:

- Peticiones: se generan de un nodo solicitante al nodo home, estas pueden ser de lectura de un bloque, de lectura con acceso exclusivo, petición de acceso exclusivo sin lectura, y petición de posescritura.

- Reenvío: los envía el nodo home a un nodo no solicitante como resultado de una

petición anterior, puede tratarse de un reenvió para informar de invalidación, o para solicitar lectura o lectura exclusiva.

- Respuestas: este puede suceder como resultado de un reenvío o de una petición, por lo que puede ser generado del nodo home al nodo solicitante, o de un nodo no solicitante al nodo home, entre este tipo de transferencias tenemos: respuesta con bloques de memoria para cargar en caché o memoria, respuesta de invalidación o respuestas del bloque invalidado.

-8.6.2 MSI con directorios con difusión

Este sistema actuará como el MSI con directorios sin difusión, pero esta vez la propagación sí se hará por difusión, lo que provocará que se envíen posiciones a todo los nodos existentes.

Así pues tendremos los siguientes tipos de nodos: nodo solicitante, nodo home, nodo modificado, nodo propietario y nodo compartidor.

Entre las transferencias encontraremos 2 grupos:

- Difusiones: generadas por un nodo tipo solicitante que enviará tanto al nodo home como a los nodo propietarios del bloque de memoria, entre estas transferencias están la petición de lectura, el petición de lectura con acceso exclusivo, petición de acceso lectura sin escritura y posescritura (esta solo se la mandará al home).

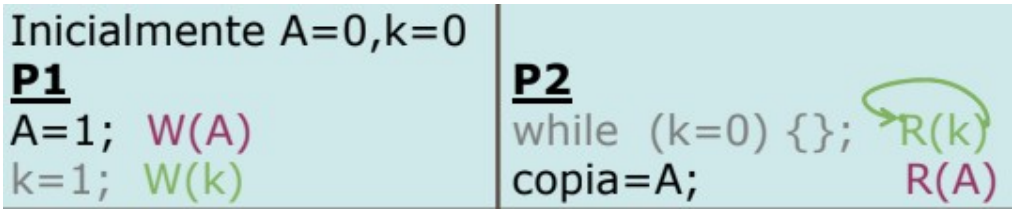
- Respuestas: estas pueden ser generadas tanto de un nodo propietario de un bloque de memoria del que piden lectura al nodo home, o del nodo home al nodo solicitante, estás transferencias son la respuesta de bloque, respuesta de invalidación o respuesta de invalidación con bloque.

(Este lección es bastante práctica, en caso de que esto se mantenga en los apuntes finales es por que he subido algunos ejercicios hechos al final de los apuntes, por lo que te recomiendo ir a mirarlos como información complementaria).

Lección 9: Consistencia del sistema de memoria

9.1 Concepto de consistencia de memoria

A la hora de trabajar de forma paralela en un sistema de memoria compartida (un multiprocesador) podemos optar subdividir un programa secuencial entre múltiple procesadores. Como es de entender, esto conlleva un claro aumento en las prestaciones pero hay que tener en cuenta una cosa, a la hora de paralelizar un programa secuencial, los compiladores no tienen porque ejecutar el código en el mismo orden que lo generamos nosotros, esto puede suceder porque los compiladores suelen tratar de organizar las ordenes que hay que ejecutar para poder evitar los riesgos a nivel de instrucción (dependencia de datos, riesgo de control o estructurales), pero esto puede causar problemas si no se tiene cuidado, he aquí un ejemplo:



Lo lógico sería pensar que en el procesador 1 primero se realizaría la función de A=1 y luego k=1, provocando que siempre se leyese el valor 1 en copia, no obstante puede suceder que el compilador decida que k=1 se ejecuta antes que A=1, por lo que podría suceder que en P2 se desbloquee el bucle while (k==0){} antes de que se haya ejecutado A=1, lo que resultaría paradójico para el programador.

Por tanto vamos a definir un modelo de consistencia de memoria como un modelo que especifica el orden en el cual las operaciones de acceso a memoria deben "parecer" haberse realizado, aunque no correspondan con el orden real que haya ejecutado el multiprocesador.

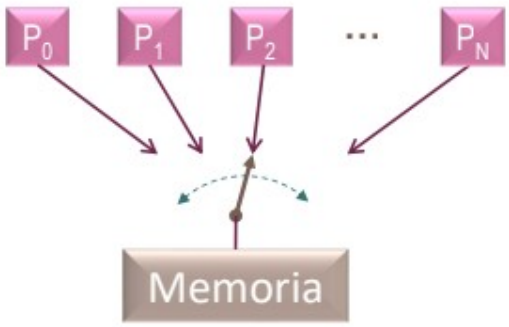
9.2 Consistencia secuencial

El modelo de consistencia de Lamport formalizado en 1979, es un modelo que dado una ejecución realizada en un multiprocesador, el resultado de esta ejecución debe coincidir con el que se obtendría de haberlo realizado de manera secuencial.

Por tanto vamos a definir los 2 requisitos fundamentales para obtener consistencia secuencial:

1. Orden del programa: En cada procesador se debe mantener el orden entre operaciones como se realizaría en el programa de forma secuencial.

2. Atomicidad: los procesadores deben actuar de forma global, de tal forma que las operaciones de memoria parezcan haberse realizado en un único procesador. Es decir, en la visión del programador, la memoria debe parecer estar conectadas a los procesadores a través de un conmutador (que no es así) y que por tanto solo permite a un procesador acceder a memoria a la vez. Así pues hasta que un procesador no haya terminado finalmente de escribir un valor en memoria, los demás procesadores no tengan acceso al valor de memoria nuevo.



Respecto al primer requisito, el objetivo es que el programador pueda intuir el funcionamiento y el resultado del programa, por ejemplo en el ejemplo dado en el apartado 9.1, el programador espera que el ordenr W⇛W (escritura con escritura) se mantenga, por lo que el resultado debería ser siempre copia=1.

Otro ejemplo sería que no se respetase el orden W⇛R, que puede darse en el siguiente ejemplo:

Inicialmente k1=k2=0

P1	**P2**
k1=1;	k2=1;
if (k2=0) {	if (k1=0) {
Sección crítica	Sección crítica
};	};

De forma general se esperaría que un solo proceso pueda acceder a la sección crítica, no obstante puede pasar que el if (k1=0) (lectura) suceda antes que el k1=1 (escritura), por lo que podría suceder que ambos procesadores se encontrasen de manera simultánea en la sección crítica.

9.3 Modelos de consistencia relajados

Pese de lo que hemos hablado anteriormente de la importancia de los modelos de consistencia lineal, a día de hoy podemos encontrar arquitecturas con relajación de consistencia, lo que implica que a favor de un aumento de las prestaciones, pueden optar por no llevar a raja tabla las ordenes de consistencia, así pues pueden relajar ambos requisitos que hemos comentado antes:

1. Orden del programa: los modelos de consistencia relajados pueden permitir alterar el orden entre dos accesos a memoria de distintas direcciones, por lo que pueden diferir tanto en la consistencias de tipo W⇛W, como en las R⇛W y las W⇛R,

2. Atomicidad: hay modelos que permiten que un procesador pueda leer el valor escrito por otro procesador antes de que esta escritura se haga visible al resto de procesadores, pongo un ejemplo de cuando esto puede suponer un problema:

P1	Inicialmente	A=B=0

A=1;	**P2**
	if (A=1)
	B=1;
		P3
		if (B=1)
		reg1=A;

Aquí cabe de esperar que al final del programa reg1 valga 1, no obstante puede suceder que el procesador P2 lea la escritura de A=1 antes de que esta se realice efectiva (si esto no afectase al resultado general sería muy bueno para las prestaciones), pero hace que B pase a 1 y que por tanto P3 pueda efectuar reg1=A, si durante todo ese rato P1 no había acabado con la ejecución de A=1, podría suceder que encontrásemos que reg1 valga 0.

En la siguiente tabla se muestran algunas arquitecturas hardware que presentan modelos de consistencia relajadas:

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		l-m-e (instruc. lectura-modificación-escritura atómica)
Sparc-PSO	Si	Si		Si		l-m-e, `STBAR` (instrucción *STore BARrier*)
Sparc-RMO	Si	Si	Si	Si		`MEMBAR` (instrucción *MEMory BARrier*)
PowerPC	Si	Si	Si	Si	Si	`SYNC`, `ISYNC` (instrucciones *SYNChronization*), LWSYNC
Itanium	Si	Si	Si	Si		`LD.ACQ`, `ST.REL`, `MF` (*ACQuisition LoaD, RELease STore, Memory Fence*),y `cmpxchg8.acq` y otras l-m-e
ARMv7	Si	Si	Si	Si	Si	`DMB` (*Data Memory Barrier*)
ARMv8	Si	Si	Si	Si	Si	`LDA` | `LDAR`, `STL` | `STLR` (*LoaD-Acquire*, *STore-reLease* 32b|64b), `LDAEX` | `LDAXR`, `STLEX` | `STLXR` (*LoaD-Acquire eXclusive*, *Store-reLease eXclusive* 32b|64b), `DMB`,l-m-e (LSE–*Large System Extension*) en ARMv8.1

(Para los que tengáis a Mancia como yo: hay que saberse para los ejercicios las cualidades de x86 y de ARMv7, por ejemplo si en un ejercicio de consistencia dicen que trabajamos con ARMv7, todo tipo de relajación de consistencias estarán presentes)

9.3.1 Modelos en los que relajan W⇒R

En estos modelos se permiten que las operaciones ejecutadas en un procesador las lecturas adelanten a escrituras, siempre que no produzcan dependencias de datos, también cuentan un buffer de tipo FIFO para las escritura en procesadores, permitiendo así que las lecturas puedan llegar a adelantar a las lecturas, debido a esto, generalmente estos sistemas permiten que el propio procesador pueda leer directamente del buffer, lo que permite leer antes que los demás procesadores una escritura propia, lo que permite leer antes que los demás procesadores una escritura propia, lo que provoca que las escrituras no afectarán inmediatamente al resto de procesadores, por tanto algunos de estos procesadores pueden optar por relajar el acceso atómico a memoria.

También quedaría comentar que para garantizar el orden correcto de los programas, estas arquitecturas suelen utilizar instrucciones de serialización, permitiendo así que aunque las lecturas adelanten las escrituras, esto no suponga un problema para ningún tipo de programa.

9.3.2 Modelos que relajan W⇒W y R⇒W

Estos modelos al igual que los anteriores, mantienen el buffer FIFO que permite adelantar lecturas frente a escrituras, pero además estos modelos permiten adelantar escrituras frente a otras, esto lo obtiene permitiendo a partir de hardware, por ejemplo

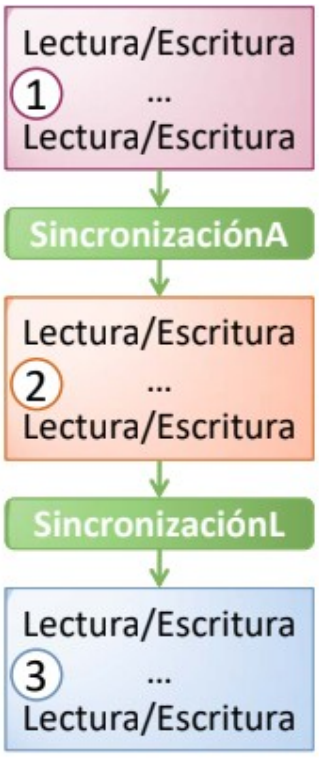
puede hacer que su red de interconexión permita que se solapen 2 escrituras en memoria o a cachés mientras se han diferentes direcciones finales. Este modelo se encuentra en los modelos tipo PSO (parcial store order) y por tanto se utilizan en los sistemas Sun Sparc.

Queda recalcar que al no mantener la consistencias con las escrituras, puede llegar a ejecutarse problemas como los explicados en el ejemplo 9.1.

9.3.3 Modelo de ordenación débil

Este modelo relaja tanto W⇛R, como W⇛W y R⇛W, este tipo de relajación se llevará a cabo siempre que esto no implique algún tipo de dependencia o riesgo, este modelo se basa en mantener el orden entre accesos solamente en los puntos de sincronización del código (secciones críticas), en el momento que se necesita coordinar el acceso a una variable compartida para los procesadores, el modelo añade código extra de sincronización. Así pues estos modelos cuenta con 2 tipos de operaciones: operaciones con datos (WR) y las operaciones de sincronización (S).

Si aparece una orden S durante una zona del código, el hardware se debe de encargar de terminar todas las operaciones tipo WR antes de completar la operación de sincronización WR⇛S, y de que esta operación S tiene que terminar antes que las operaciones WR que prosiguen a esta.



9.3.4 Modelo de consistencia de liberación

Este modelo deriva del de ordenación débil pues tiene en cuenta que hay adicionalmente dos tipos de código para sincronización, tendrá pues la operaciones de adquisición (SA), que serán las que efectuarán los procesos para ganar acceso único a variables o recursos compartidos, y las operaciones de liberación (SL), que se utiliza para que un procesador le permita a otro el acceso a recursos o variables que compartan.

Por tanto, en una zona de sincronización, se debe asegurar que hay que mantener el orden entre una operación de adquisición y cualquier orden posterior SA⇛WR, y mantener el orden entre cualquier operación de datos y una operación de liberación posterior WR⇛SL.

Lección 10: Sincronización

10.1 Comunicación en multiprocesadores y necesidad de usar código de sincronización

La comunicación entre procesos en multiprocesadores es vital a la hora de trabajar con estos, pero toca recordar que esta comunicación se hace a partir de memoria compartida, así pues para poder implementar de forma efectiva la comunicación entre procesos, se necesita sincronizar el acceso a las variables compartidas, esta comunicación se necesita tanto en la comunicación uno a uno como en el de una comunicación con múltiples procesos involucrados (lo que llamamos comunicación colectiva):

-Comunicación uno a uno: si tenemos 2 procesos, uno encargado de escribir valores en una memoria compartida y otro encargado de leerlos con el fin de trabajar con ellos, es imprescindible sincronizar los procesos con el objetivo de que o se acceda a los valores antes de haberse escrito en ellos, o en caso de que se reutilicen esos valores, de que se sobrescriban antes de que se haya leído su valor anterior. Es decir, necesitamos algún mecanismo que limite el acceso de un solo procesador a una región de memoria compartida durante algunas instrucciones de una ejecución, la que denominaremos sección crítica o zona de exclusión mutua.

-Comunicación colectiva: en este caso la zona de exclusión mutua se generará debido a lo que conoceremos como condición de carrera, la condición de carrera está presente cuando tenemos múltiples procesadores escribiendo sobre una misma variable, veamos el siguiente ejemplo:

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

Race condition

El problema de este código reside en la línea sum=sum+sump, pues puede suceder que dos procesadores P1 y P2, con lo valores 2 y 3 en sump respectivamente, y vayan a realizar la escritura de forma simultanea, para ello primero cargarían el valor de sum, que supongamos que en ese momento vale 1, así que P1 escribiría en sum 1+2, lo que en principio sería correcto, pero P2 ya tenía cargado que sum era 1 previamente, por lo que en vez de tener 3, mantendría el 1, por lo que haría 1+3.

Smilke®

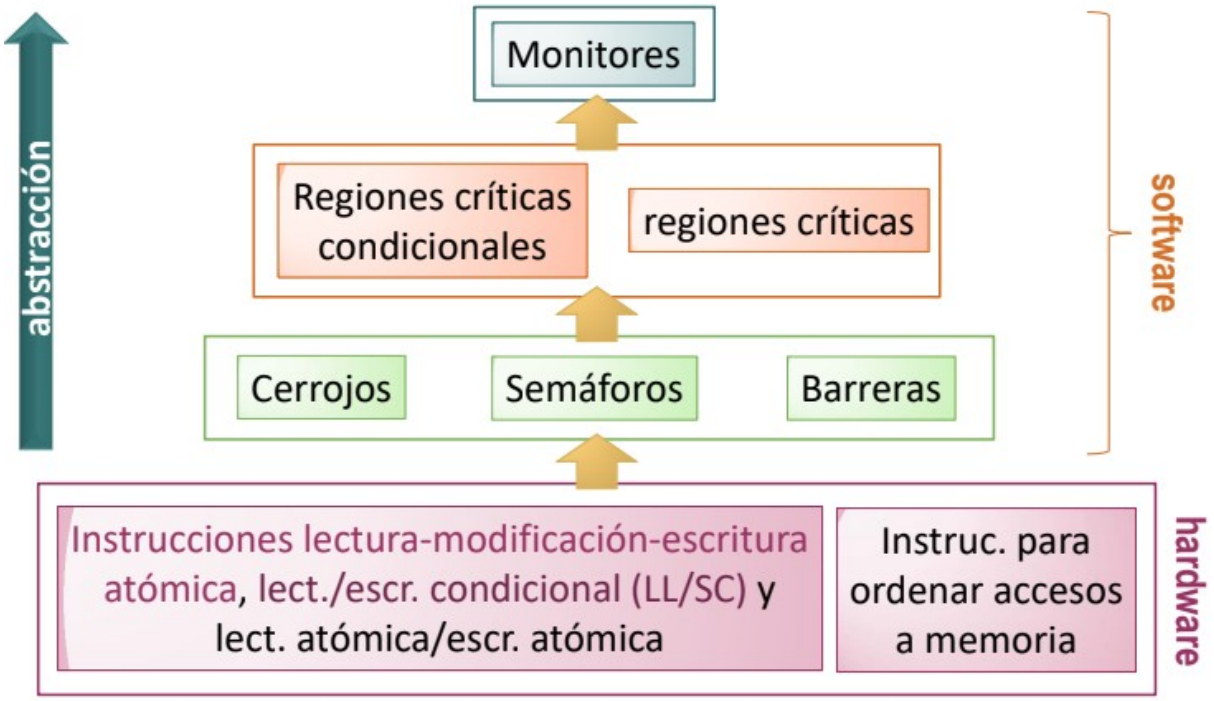
PARA ESTE EXAMEN AÚN TE FALTAN UN PAR DE CAFÉS MÁS. DISFRÚTALOS COMO SE MERECEN.

Descuentazo para tí

CÓDIGO: WUOLAH10

COMPRA AQUÍ

A la hora de dar soporte a la sincronizacion, podemos encontrar algunos soportes tanto hardware como software, algunos de ellos serán tema de estudio más adelante.



10.1 Cerrojos

Los cerrojos son métodos que proporcionan una forma de asegurar exclusión mutua, para ello definen 2 funciones para sincronizar:

- Cierre del cerrojo o lock(k): esta función un proceso intenta adquirir el derecho de acceder a una sección crítica, y solo un proceso puede tener ese acceso, así pues todos los procesos que intenten adquirir el cerrojo estando este ya cerrado, entrarían en una etapa de espera hasta que se desaloje el cerrojo.

- Apertura del cerrojo o unlock(k): esta función libera a uno de los procesos que esperan el acceso a la sección crítica, o en caso de no haber procesos en espera, permitir que el próximo proceso que realice una llamada a lock, se le acceda el acceso a la zona crítica,+.

10.1.1 Componentes fundamentales para la creación de un cerrojo

Así pues una vez entendido las forma fundamental de un cerrojo, antes de profundizar en estos, vamos a repasar que componentes fundamentales debe tener uno de estos:

- Método de adquisición: este es el método que relega el derecho a un proceso de poder acceder a la zona de exclusión mutua. Algunas arquitecturas como x86 utilizan instrucciones de lectura-modificación-escritura atómica para implementarlos, y otros como ARMv7 utilizan carga enlazada con almacenamiento condiciona (LL/SC)

(ambos métodos los daremos más adelante).

- Método de espera: este es el método encargado de que hacer esperar a un thread hasta que este pueda acceder a la zona de exlución mutua. Para ello se utilizan 2 métodos principales:

    *Espera ocupada: consiste en hacer que el thread dedique en bucle a mirar si el recurso está desocupado, y hasta que no lo esté no podrá salir de ahí.

    *Bloque: el sistema operativo desaloja al proceso de la cpu hasta que el recurso este disponible.

- Método de liberación: estos métodos se usan para señalar a los threads en estado de espera de que el recurso que quieren obtener está desalojado, en los casos de los cerrojos este método solo libera a un thread, pero veremos en próximos apartados, que en las barreras se liberan múltiples hebras a la vez.

10.1.2 Cerrojo simple

La implementación de estos cerrojos se aplicarán a través del uso de una variable compartida k que podrá tomar 2 valores: 0 para abierto y 1 para cerrado.

A la hora de crear la función unlock(k), lo único que tendremos que hacer para desalojar será escribir a 0 el valor de k.

La función lock es un poco más compleja, pues lo primero que tendremos que hacer es ver si el valor de k es 0 o 1, en caso de ser 1 deberá acceder al método de espera que hayamos proporcionado, pero si es 0 este deberá escribir en el valor de k para que pase 1, y ya finalmente pase a la zona crítica. El problema principal radica que la lectura de k y su correspondiente cambio a 1 si es 0 debe realizarse de forma **ATÓMICA** pues si no se realizase así podría suceder que 2 o más hebras realizasen la lectura a la vez y provocar que más de una hebra se metan en la zona de exclusión mutua.

lock (k)

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

unlock (k)

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

(Para los que tengáis a Mancia como yo: ha avisado que en los exámenes utiliza cerrojos de openMP, que son los que damos en prácticas, echadle un vistazo para familiarizaros de como son para que no os pille desprevenido en los exámenes)

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)`

10.1.3 Cerrojos con etiquetas

Un problema fundamental que tiene los cerrojos simples es lo que se llama inanición de procesos, esto sucede cuando un proceso queda en estado de espera, pero a la hora de que se libere el recurso por el que espera, se le adelanten otros procesos a la hora de quedarse con este, provocando que el proceso quede forma infinita en estado de espera. Para solucionar este problema se inventaron los cerrojos con etiquetas, estos en vez de tener 2 valores para poder entrar o no a la zona de exclusión mutua, si no que se implementará una cola FIFO con 2 punteros, los procesadores que soliciten entrar a la sección crítica, obtendrán la posición del puntero que se encarga de los procesos bloqueados, y este puntero se moverá hacia la derecha en la cola, el proceso quedará esperando hasta que el otro puntero, encargado de la liberación, señale a la posición del valor que había copiado en su momento del puntero de bloqueados.

Una vez que un proceso termina su estancia en la sección crítica, llamará a la función

unlock, que se encargará de desplazar el puntero de liberación a la derecha.

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

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

10.2 Barreras

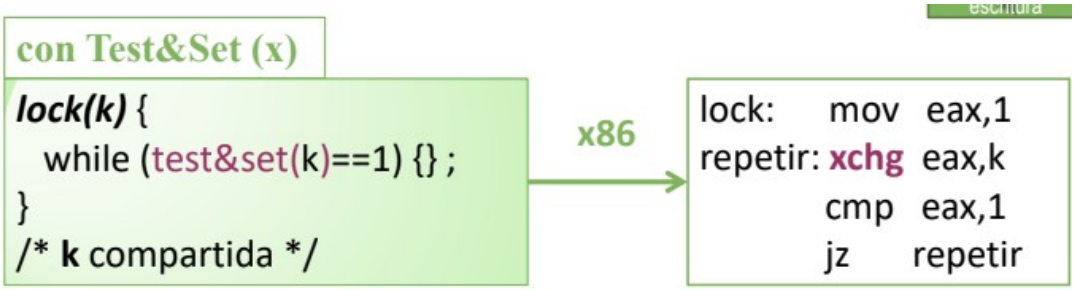
Estas funciones se utilizan para sincronizar entre sí, en algún punto de código, los procesos que colaboran en la ejecución de un trozo de código, las barreras se encargan de que ningún proceso pueda sobrepasar un punto de código hasta que todos los procesos hayan llegado a ese punto. Esto típicamente se utiliza para separar fases del código paralelo en los que nos interesan que todos los procesos hayan terminado su trabajo final.

10.3 Aporte hardware a primitvas software

Como hemos visto antes, hay un problema a la hora de generar los métodos de adquisición a la hora de realizar un software de sincronización, y es que necesitamos que esta adquisición debe trabajar de forma atómica para evitar que varias hebras entre en una sección crítica de manera simultánea, por suerte contamos con algunos mecanismo software que nos permiten evitar este problema.

10.3.1 Función Test&Set

Esta primitiva es muy simple, se le pasa una variable compartida x, y lo que hace es guardar en un registro local el valor de x, procede a asignarle el valor 1 a la variable x y devuelve el valor del registro local, esto permite tanto comprobar si el valor es 0 o 1, y de forma automática también escribir un 1 en ese valor, esto permite generar fácilmente la función lock de un cerrojo sin tener problemas de no atomicidad como hablé en el 10.1.2:



Este cerrojo tendrá el valor 0 para cerrado y el valor 1 para abierto, así pues si un proceso llega a la función lock cuando k vale 1, este simplemente entrará en el estado de espera utilizando espera ocupada, en caso de que fuese 0, el valor de k pasaría a 1 de forma atómica pero aún así se devolvería 0 en la función, por lo que podría acceder a la zona de exclusión mutua.

Importante remarcar que en este tipo de directivas vamos a trabajar con arquitectura x86, y por tanto contamos con la instrucción en ensamblador xchg que es lo que realmente permite intercambiar un valor de memoria y el valor de un registro de forma atómica.

10.3.2 Primitiva de intercambio

Esta primitiva no cuenta con una función fija como en los otros casos que estamos dando debido a que es la aplicación del uso de la instrucción xchg, que nos permite generalizar una función como la de test&set pero con cualquier valor, para aclaralo voy a escribir lo fácil que sería en ensamblador aplicar este lock con el valor 2 en vez de, que es como funciona test&set:

```
lock:       mov eax,2
repetir:    xchg eax,k      (Exacto, es el mismo código que test&set
            cmp eax,2             cambiando 1 por  2)
            jz repetir
```

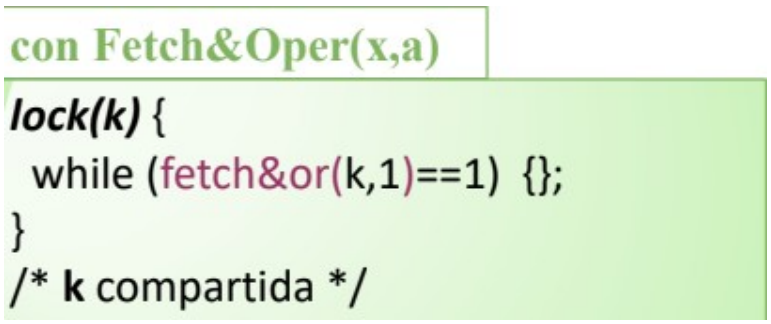
10.3.3 Primitiva Fetch&Operation

Test&set era muy simple pero siempre trabajaba con el valor 1, fetch and operation es otra primitiva hardware que aunque más compleja, da una mayor flexibilidad a la implementación de la función lock de manera atómica.

Esta directiva tiene múltiples formas pues la parte de Operation dentro del nombre es una referencia a una operación de tipo atómico con el que trabajará la operación, por lo que podremos encontrar funciones como Fetch&Or Fetch&Add o

Fetch&Increment.

Esta directiva cuenta con 1 o 2 parámetros, dependiendo de la operación de la instrucción fetch, el primer parámetro hará referencia a la variable compartida con la que trabajará el cerrojo, el valor de esta se guardará en un registro temporal tal y como sucedia en el test&set, luego se producirá la operación especificada entre k y la segunda variable que se le pasa (si se requiere de segunda variable) y el resultado se guardará en k, finalmente se devolverá el valor del registro temporal. Veamos este ejemplo:



¿Cómo funciona esta función lock? Bien la función fetch&or lo que hará será comprobar y devolver el estado de k y sobrescribir el valor de k a 1 (pues un or de cualquier valor con 1 siempre da 1), y luego comprueba si el valor que devuelve la función es 1. Pues a través de eso podemos deducir que la funcionalidad de este cerrojo es ver que si k es 1, se mantendrá en estado de espera pues entrará en el bucle while, por otro lado si el valor de k fuese 0, este pasaría a 1, pero la función devolvería 0 así que no solamente bloquea el paso del resto de procesos, sino que además permite saltarse el bucle while.

Finalmente queda añadir que esta función se implementa con la instrucción x86 llamada lock xoperation (por ejemplo lock xadd) que lo que hace de forma atómica es guardar el intercambio de memoria registro, guardando realmente en memoria la operación que hemos especificado:

LOCK XADD x,y haría (y=x y x=x+y) de forma simultánea.

10.3.4 Primitiva Compare&Swap

Esta primitiva también es parecida a las otras 3 que hemos dado anteriormente, esta cuenta con 3 parámetros, los 2 primeros son valores locales, a y b, y el otro la variable k, que en como las otras primitivas, es la variable de memoria compartida con la que trabajará el cerrojo. Lo que hará la primitiva es muy simple, de forma atómica comprobará si el valor de k y a coinciden, en caso de ser cierto los valores k y b se intercambiarán, veamos el siguiente ejemplo:

con Compare&Swap(a,b,x)

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

Este ejemplillo es muy simple, creamos una variable b que se iniciará a 1, y se procederá a comprobar si el valor del cerrojo k es 0 o no, en caso de que no sea así, b seguirá en 1, provocando que el proceso quedase en estado de espera ocupada, en caso de k fuese 0, este intercambiaría el valor con b pasando a ser 1, consiguiendo así que los demás procesos quedasen bloqueados.

En x86 para realizar esta función de forma atómica necesitaremos de la instrucción LOCK CMPXCHG mem, reg, que lo que hará será comprobar si el valor de mem y el registro eax coinciden, de ser así mem y reg se intercambiarán, y todo esto de forma atómica claramente.

(Para los de Mancia: las malas lenguas me han contado que en algunos exámenes ha llegado a poner exámenes con código en ensamblador, no sé hasta que punto es esto cierto, pero no estaría demás que os aprendieseis las ordenes en ensamblador que he explicado por si las moscas).

10.3.5 Primitiva de carga enlazada con almacenamiento adicional (LL/SC)

Esta primitiva realmente no es una, sino que se compone de 2 instrucciones que van en secuencia y juntas conforman una primitiva que impide que 2 procesadores entren de forma simultánea en la sección crítica.

Estas dos instrucciones (que se suelen utilizar en arquitecturas como ARMv7) son LL (load linked) y SC (store conditional). LL lo que hace es enlazar una posición de memoria que se carga en un procesador, de tal forma de que si el valor de esa posición de memoria cambia antes de que el procesador pueda realizar el guardado condicional (SC), este no realice el almacenamiento de ese dato. Esto aunque no atómico, evita que dos procesos puedan entrar en la zona de exclusión mutua a la vez.