

1. Introducción

3 tipos de paralelismo:

- Paralelismo a nivel de datos (DLP) --> una misma instrucción sobre múltiples datos.
- Paralelismo a nivel de instrucciones (ILP) --> ejecutar múltiples instrucciones al mismo tiempo dentro de un mismo hilo. Implementado por procesadores superescalares o segmentados (pipelines), donde varias unidades trabajan sobre instrucciones de una misma secuencia.
- Paralelismo a nivel de hebra (TLP) --> ejecutar varios hilos en paralelo. Pueden ser de diferentes flujos de instrucciones o programas.

TLP nace por la necesidad de las arquitecturas para mejorar el rendimiento. Aumenta el rendimiento global.

2.

Multiprocesadores y multicores (TLP con varios procesadores)

Un multiprocesador (varios procesadores en un mismo computador y SO compartiendo memoria principal) forman un SMP (Symmetric Multiprocessor).

SMP clásico: acceso a memoria uniforme (UMA) para todos, misma latencia para todos.

SMP más grandes: acceso NUMA, donde cada procesador/núcleo tiene una parte de la memoria reservada como local (acceso más rápido), mientras que el resto es acceso remoto (más lento).

Un sistema multiprocesador ejecuta hilos distintos en paralelo. Si están en el mismo chip, hablamos de multicore.

Si hablamos de cluster donde hay varios equipos con SO distintos, estos conocimientos no aplican ya que el paralelismo se gestiona via comunicación (MPI, mensajes).

3.

Cores multihebras

Se puede dar TLP dentro de un mismo núcleo.

Un core multihebra diseñado para mantener varios hilos hardware y compartir la ejecución entre ellos. Hay 3 tipos de enfoques:

- Multithreading de grano fino --> en cada ciclo de reloj se ejecuta un hilo distinto, evitando así que un solo hilo bloquee el pipeline. Requiere hardware capaz de hacer estos cambios rápidos. El avance de los hilos es más lento.

- Multithreading de grano grueso --> procesador ejecuta hilo durante varios ciclos seguidos, y cambia en caso de eventos de larga latencia o intervalos de tiempo.

Mantiene una mejor cache para el hilo que corre. Puede implicar algunos ciclos perdidos debido a la hora de hacer los cambios.

- Multithreading simultáneo --> los hilos realmente se ejecutan en paralelo en un mismo ciclo. El procesador emite instrucciones de varios hilos en un mismo ciclo. Puede provocar competiciones entre hilos para

consumir los recursos internos como cache.

4.

Coherencia de memoria en multiprocesadores

Al compartir memoria entre varios procesadores puede haber problemas de coherencia de cache.

Hay que mantener la integridad de los datos: todas las copias de las caches que apunten a una misma direcci3n deben reflejar el mismo valor. Si un procesador modifica un dato, los dem3s deben enterarse de ese cambio.

Requisitos para coherencia de cach3:

1. Propagaci3n de escrituras --> si un procesador escribe x en su cache, tarde o temprano ninguna otra cache podr3 no tener ese valor en su copia

2. Serializaci3n de escrituras --> si dos escrituras distintas ocurren en la misma direcci3n, todos los procesadores deben verlas en el mismo orden (aunque no sea un orden exacto se establece un orden global coherente). Esto garantiza un 3nico valor 3ltimo para cada direcci3n, evitando condiciones de carrera.

Resumen: para cada posici3n de memoria (cada direcci3n), los valores se actualizan de manera consistente en todo el sistema.

Situaciones de incoherencia:

- CPU0 escribe en su cach3 un dato X=5 (antes X=0), pero CPU1 a3n tiene X=0 en su propia cach3 3 si CPU1 lee X, obtendr3 un valor

obsoleto.

- Dos CPU suman al mismo contador en paralelo: sin coherencia, cada uno podríá cachear el valor inicial y actualizar localmente, sobrescribiendo el resultado del otro (problema de actualización perdida).

Para resolver estas cosas, los sistemas multiprocesadores incorporan protocolos de coherencia de caché:

- Protocolos de "snooping" (espionaje) --> cada caché se entera (espiando señales) cuando otro núcleo hace una petición o escritura a una dirección que dicha caché almacena.

Al detectarla, la caché puede actualizar el nuevo valor si se difunde o invalidar su copia (al saber que puede estar obsoleta). Estos protocolos suelen necesitar una interconexión difusión/broadcast.

- Protocolos basados en directorios --> hay una estructura centralizada llamada directorio, que lleva un registro de bloques administra cada caché. En caso de querer modificar uno, se coordina con dicha caché, por ejemplo mediante mensajes. Esto escala mejor en sistemas con muchos procesadores al evitar el broadcast masivo

- Write-Invalidate --> ante una escritura, bloquear automáticamente copias ajenas, de modo que quien escribe sea el único con una copia válida

- Write-Update --> ante una escritura, se envía el nuevo valor para una actualización masiva.

4.2

Ejemplo de Coherencia (Protocolo MESI en acción)

Escenario: Consideramos 4 procesadores (P1-P4), cada uno con caché propia conectada mediante un bus común (SMP). Inicialmente, ninguna caché contiene el bloque X (todas en estado Invalido, I), pero la memoria principal contiene el valor actualizado del bloque X. Veamos cómo el protocolo MESI mantiene la coherencia mediante una secuencia de operaciones:

P1 lee dirección X:

P1 no tiene X, genera una Petición de Lectura (PtLec) en el bus.

La memoria responde con el bloque X, que P1 almacena en caché en estado Exclusivo (E), indicando que es la única copia válida y coincide con memoria.

Estado tras operación: P1=E, P2=I, P3=I, P4=I.

P2 lee dirección X:

P2 tampoco tiene X, genera otra PtLec.

Ahora el bloque está en P1 en estado E (no modificado), la memoria principal también está actualizada. Puede responder la memoria o P1 directamente (depende del diseño específico).

Tras esta operaci3n, P1 y P2 comparten el bloque en estado Compartido (S).

Estado actual: P1=S, P2=S, P3=I, P4=I.

P1 escribe direcci3n X:

Para escribir, P1 necesita exclusividad, genera una Petic3n de Lectura Exclusiva (PtLecEx).

Todas las dem3s cach3s que tengan copia de X (P2 en este caso) invalidan su copia pasando a estado Inv3lido (I).

P1 recibe confirmaci3n, cambia su bloque a Modificado (M) y realiza la escritura, dejando memoria temporalmente desactualizada.

Estado actual: P1=M, P2=I, P3=I, P4=I.

P2 escribe direcci3n X:

P2 quiere escribir X, actualmente en estado I, por lo que tambi3n lanza una PtLecEx.

P1 (en estado M) escucha esta petici3n, realiza un write-back del bloque modificado a memoria o lo envi3a directamente a P2.

P1 invalida su copia (I), P2 adquiere el bloque en estado M para su

propia escritura.

Estado actual: P1=I, P2=M, P3=I, P4=I.

P3 escribe dirección X:

P3 realiza una `PtLecEx`, solicitando exclusividad.

P2 (M) realiza flush (write-back) del bloque actualizado a memoria o directamente a P3.

P2 pasa a I y P3 obtiene el bloque en M para modificarlo.

Estado final: P1=I, P2=I, P3=M, P4=I.

El protocolo MESI asegura que solo una caché tiene una copia modificable (M) del bloque. Las demás cachés deben invalidar sus copias al detectar intentos de escritura de otros procesadores, garantizando la coherencia.

Problemas de rendimiento:

Los cambios de estados generan tráfico adicional en el bus.

La escritura concurrente frecuente sobre un mismo bloque produce efectos negativos como el "ping-pong" (invalidación continua), aumentando el tráfico.

El "falso compartido" ocurre cuando varios procesadores acceden a variables distintas dentro del mismo bloque de caché, generando invalidaciones innecesarias.

5.

Consistencia de Memoria (Modelo de Memoria)

La coherencia maneja actualizaciones de un solo dato en distintas cachés. En cambio, la consistencia define en qué orden global se observan todas las operaciones sobre distintas posiciones de memoria realizadas por varios procesadores.

Consistencia Secuencial (SC):

Operaciones de cada hilo respetan el orden programado.

Todos los procesadores ven las operaciones como intercaladas en un orden global coherente.

Ejemplo: Si un hilo escribe primero $X=5$ y luego $Y=8$, otro hilo nunca ve primero $Y=8$ antes que $X=5$.

Diferencias clave:

Coherencia: gestión de actualizaciones individuales por dirección.

Consistencia: define el orden global de todas las operaciones.

Modelos de consistencia relajados permiten optimizaciones de rendimiento:

Total Store Order (TSO): común en arquitecturas x86, permite reordenar lecturas/escrituras bajo ciertas condiciones.

Weak Consistency (débil): exige sincronización explícita.

Release Consistency: distingue operaciones de adquisición y liberación de recursos (locks).

Modelos de alto nivel (C++, Java): aún más relajados, requieren sincronización explícita con `volatile` o `atomic`.

Ejemplo:

Dos hilos: (H1: A=1, imprime B) y (H2: B=1, imprime A).

Bajo SC, no es posible que ambos impriman 0.

Bajo modelos relajados, sin sincronización explícita, sí podría ocurrir (impresión anticipada antes que propaguen escrituras).

Conclusión:

SC es idealmente sencillo pero costoso en rendimiento. Por ello, la mayoría de procesadores reales implementan modelos relajados, delegando la responsabilidad de sincronización al software (fences,

locks).

6.

Sincronización en Sistemas Multiprocesador

Para coordinar múltiples hilos paralelos se emplean mecanismos de sincronización:

Instrucciones atómicas (Test-and-Set, Compare-and-Swap, Fetch-and-Add): ejecutan operaciones indivisibles, ideales para exclusión mutua.

Spin-locks: repetición continua de test-and-set hasta adquirir recurso. Consumo activo de CPU, pero sencillo de implementar.

Fences (barreras de memoria): instrucciones explícitas (MFENCE en x86) garantizan orden de memoria, imprescindibles en modelos relajados.

Semáforos, mutex, monitores: herramientas de alto nivel basadas en instrucciones atómicas, bloquean y despiertan hilos mediante planificador del SO.

Barreras de sincronización: sincronizan múltiples hilos en puntos específicos, utilizando contadores atómicos y banderas globales.

Sin una correcta sincronización pueden ocurrir condiciones de carrera,

con resultados no determinísticos e incorrectos. Por ello, se deben emplear siempre las primitivas atómicas y mecanismos del hardware/software adecuados.