Subsistema de memoria

Organización de computadoras 2015

Universidad Nacional de Quilmes

1 Subsistema de memoria

El sistema de computos necesita tener varios tipos de memorias, para ser utilizadas en diferentes situaciones, cumpliendo distintos objetivos. Antes de mostrar porque es esto cierto, veamos cómo caracterizar las memorias según los diferentes aspectos.

En primer lugar, una unidad puede ser interna o externa al sistema según si es fija o desmontable (esto es, portable a otro sistema).

Además puede ser volátil, si su contenido se pierde al perder la alimentación eléctrica, o persistente, si no necesita electricidad para mantener la información.

Puede ser de lectura y escritura, como se requiere en una memoria principal, o bien de solo lectura para almacenar información estática que no necesita modificarse.

Por último, se las puede clasificar segun su método de acceso, en secuencial, directo, aleatorio o asociativo. El **método de acceso secuencial** requiere que la dirección (o identificación) de cada dato esté almacenada junto con él, y por lo tanto el dispositivo debe recorrerse secuencialmente hasta encontrar la identificación buscada. El **método de acceso directo** es el utilizado por los discos magnéticos, donde la direccion del dato se basa en su ubicación física, en particular la busqueda se da en dos etapas: se accede primero a la zona que incluye al dato y luego se busca secuencialmente dentro de esa zona. En el **método aleatorio** cada dato tiene un mecanismo de acceso único y cableado físicamente (cada acceso es independiente de los accesos anteriores). Finalmente, el **método asociativo** organiza cada unidad de información con una etiqueta que la describe en función de su contenido. Entonces, para recuperar un dato se debe analizar un determinado conjunto de bits dentro de la celda, buscando la coincidencia con la clave o patrón buscado. Esta comprobación del contenido de las celdas se lleva a cabo de manera simultánea en todas las celdas.

1.1 Memorias ROM

Para algunas aplicaciones, el programa no necesita ser modificado y entonces puede ser almacenado de manera permanente en una memoria de solo lectura ó ROM (*Read Only Memory*). Ejemplos de memorias ROM pueden encontrarse en videojuegos, calculadoras, hornos de microondas, computadoras en automóviles, etc. Además casi todas las computadoras personales necesitan una memoria ROM donde almacenar el primer programa que da arranque al sistema operativo a partir del acceso (en la mayoria de los casos) a un disco rígido primario.

1.2 Jerarquía de memorias

Como se dijo, la **memoria principal** es volátil y de acceso aleatorio. Si es volátil entonces se necesita otra posibilidad de almacenamiento donde los programas puedan almacenarse de manera persistente y desde donde se recuperen bajo demanda del usuario (cuando dispara la ejecución de un programa).

Esta **memoria secundaria** puede ser resuelta con un disco rígido o un disco de estado sólido (SSD) donde se mantengan persistentes (instalados) los programas. Cuando se necesita de la ejecución de un programa, hecho que puede darse a partir del requerimiento de un usuario o por invocación de otro programa, este debe ser cargado en memoria y permanece allí hasta que la memoria se sobreescribe o se apaga el sistema.

La pregunta que puede surgir es ¿porque no montar la memoria principal en una tecnología persistente, como puede ser un disco de estado sólido? Para responderla es importante destacar que existe una relación de compromiso entre el tiempo de acceso, el costo por bit y la capacidad de almacenamiento (ver figura 1). Un disco tiene mayor capacidad (y por lo tanto menor costo por bit) pero un tiempo de acceso mucho mayor. Esto impacta directamente en el desempeño de la CPU pues el tiempo de ejecución de una instrucción está condicionado por el tiempo de acceso a memoria principal. Además, en este punto es importante notar que algunos programas pueden no ser ejecutados nunca y algunos datos pueden no ser accedidos, aunque se los tenga disponibles en el sistema.

Figure 1: Relacion de compromiso

Es por esto que se incorpora una memoria mas pequeña y rápida (de acceso aleatorio) donde se cargan los programas a la hora de ser ejecutados y los datos cuando se los

necesita. Si lo que se necesita es asegurar una velocidad aceptable y no se necesita gran capacidad, ¿Porqué no implementar la memoria principal con regitros de la CPU? Para responder esto se debe tener en mente que el costo sea razonable. En general las arquitecturas cuentan con pocas decenas de registros debido al costo que eso implica. Por otro lado, si se implementa la memoria principal con una RAM se otorga flexibilidad para extender su tamaño pues se trata de un componente externo a la CPU y en cambio los registros suelen estar definidos en el diseño electrónico de la misma. Este esquema se describe en la figura 2.

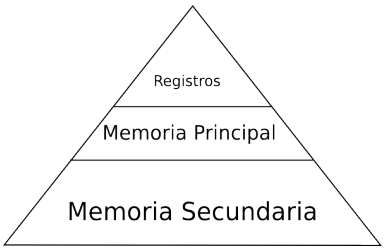

Figure 2: Jerarquía de memoria

2 Memoria Caché

Desde hace un par de décadas, el progreso tecnologico de las computadoras personales esta marcando una tendencia a duplicar, cada año y medio, la velocidad de los procesadores y el tamaño de la memoria principal pero la velocidad de las memorias apenas crece un 10%. Esto ocasiona un crecimiento de la brecha entre la velocidad del procesador y la velocidad de la memoria, causando un mayor impacto en el tiempo de ejecución de los programas, pues la velocidad con la que la CPU puede ejecutar instrucciones está limitada por el tiempo de acceso a memoria, dado que al menos una vez es necesario accederla dentro del ciclo de ejecución de instrucción.

Un enfoque para buscar una solución a esta brecha es incorporar una memoria mas pequeña que la memoria principal pero mas rápida, teniendo en mente que no es viable construir la memoria principal a partir de registros internos a la CPU. Esta idea se basa en que los accesos que se solicitan no son al azar, sino que respetan una lógica que tiene relación con la naturaleza de la ejecución de los programas, pudiendose distinguir dos tipos de accesos: lectura de instrucciones y lectura de datos. La lectura de instrucciones es, en su mayoría, un recorrido de celdas consecutivas de memoria, y asi mismo la lectura de los datos de un arreglo también lo es. Por otro lado, las instrucciones de una rutina o de una repetición se utilizan mas de una vez. Este patrón de acceso se describe con los **principios de localidad**: el principio de localidad espacial enuncia que las posiciones de memoria cercanas a alguna accedida recientemente son mas probables de ser requeridas que las mas distantes, y el principio de localidad temporal dice que cuando un programa hace referencia a una posicion de memoria, se espera que vuelva a hacerlo en poco tiempo.

A partir de esta idea, se busca tener las celdas mas usadas (y no todas) en una memoria mas inmediata, intermedia a la cpu y la memoria principal (ver figura 3), denominada **Memoria Caché**.

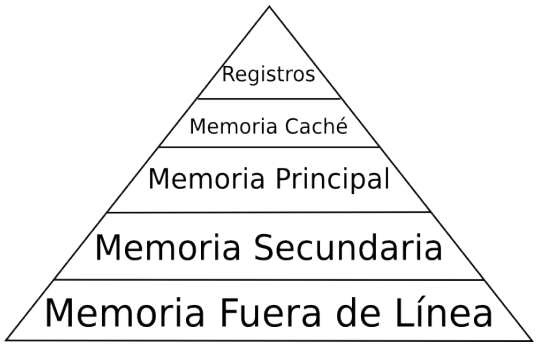

Figure 3: Jerarquía con Memoria Caché

La memoria caché tiene como objetivo proveer una velocidad de acceso cercana a la de los registros pero al mismo tiempo proveer un mayor tamaño de memoria. Para esto, la caché contiene una copia de porciones de la memoria principal, y en el momento de leer una celda, primero se verifica si se encuentra en la caché. **Si esto no ocurre** debe traerse de la memoria principal un bloque de celdas a la caché y luego la celda se entrega a la CPU. El principio de localidad dice que cuando un bloque de datos es traído a la caché porque se necesitó una de sus celdas, entonces es probable que próximamente se necesiten otras celdas del mismo bloque.

3 Funciones de correspondencia

Las funciones de correspondencia proponen distintos criterios para corresponder los bloques de memoria principal con las líneas de memoria caché, es decir: con que flexibilidad se almacenan los bloques en las líneas.

El enfoque mas natural es la **correspondencia completamente asociativa**, donde cada línea de caché se puede llenar con cualquier grupo de celdas de la memoria principal, y por lo tanto se requiere que el tag que se almacena permita identificar al bloque que acompaña. Por ejemplo, supongamos una memoria principal con direcciones de 16 bits y una memoria caché donde cada línea puede almacenar un bloque de 16 celdas. Entonces las celdas con direcciones 0000 a 000F corresponden al mismo bloque, y por lo tanto el número de bloque es 000. Las celdas 78A0 a 78AF corresponden al bloque 78A. De aqui que el tag tenga 12 bits, para almacenar el número de bloque. Para verificar si una celda está cacheada, la memoria caché controla si el número de bloque correspondiente a esa celda (ya que el numero de bloque es parte de la dirección recibida) es alguno de los tags almacenados.

La implementación de una correspondencia asociativa tiene un costo elevado debido a que se requiere la búsqueda de la clave (número de bloque) en todas las líneas en forma simultánea y esto requiere una tecnología de ac-

ceso aleatorio. Para reducir este costo es posible presentar otro mecanismo de correspondencia donde cada bloque de memoria principal esté asignado a una línea determinada. Como la cantidad de líneas es mucho menor a la cantidad de bloques (porque en otro caso la memoria principal y la caché tendrían la misma capacidad), hay un conjunto de bloques candidatos para una misma línea que "compiten" entre si. Este mecanismo se denomina **correspondencia directa** y cada línea de caché es un recurso a compartir por un conjunto de bloques de memoria. Es importante marcar que este conjunto de bloques no es consecutivo, con el objetivo de aprovechar el principio de localidad y maximizar la tasa de aciertos. A la hora de verificar si una celda está cacheada, la memoria caché compara solamente en una línea (la que corresponde al bloque) para determinar si contiene el tag buscado, simplificando la tecnología que se necesita para implementar la caché.

Por ejemplo, si se analiza la misma memoria principal y caché del ejemplo anterior, aplicando en cambio una correspondencia directa, se obtienen 2^{12} bloques a asignar entre las líneas de caché. Supongamos 64 líneas de caché, entonces le corresponde a cada una $2^{12}/2^6 = 2^6$ bloques. Por lo tanto, el tag debe distinguir esos $2^6 = 64$ bloques posibles, y de aqui que tenga 6 bits de tamaño.

A la hora de comparar los enfoques anteriores, se ve que la correspondencia directa es mas económica en su construcción pero la correspondencia asociativa es mas flexible y maximiza el porcentaje de aciertos. Para hacerlo evidente, suponer una secuencia de accesos que requiera repetidamente cachear bloques que corresponden a una misma línea, causando repetidos fallos. En una correspondencia asociativa, esos bloques no compiten y no se darían mas fallos que los que producen bloques nuevos.

En una búsqueda de balancear los dos aspectos mencionados (eficiencia vs. costo), se propone una **correspondencia asociativa por conjuntos**, donde la memoria caché se divide en conjuntos de N líneas y a cada bloque le corresponde uno de ellos. Es decir que dentro del conjunto de líneas asignado, un bloque de memoria principal puede ser alojado en cualquiera de las N líneas que lo forman, es decir que dentro de cada conjunto la caché es totalmente asociativa.

Esta situación es la más equilibrada, puesto que se trata de un compromiso entre las técnicas anteriores: si N es igual a 1, se tiene una caché de mapeo directo, y si N es igual al número de líneas de la caché, se tiene una caché completamente asociativa. Si se escoge un valor de N apropiado, se alcanza la solución óptima.

4 Políticas de reemplazo

Cuando se produce un fallo y un nuevo bloque debe ser cargado en la caché, debe elegirse una ranura que puede o no contener otro bloque, para ocupar. Para el mapeo directo hay solo una posible ranura para un determinado bloque, de modo que no hay eleccion posible. Sin embargo, para los mapeos asociativo y asociativo por conjuntos se necesita un algoritmo de reemplazo.

Menos recientemente usado (LRU)

La política LRU (por sus siglas en inglés: *Least Recently Used*) reemplaza el bloque que ha estado mas tiempo en la caché sin ser usado, es decir, sin referencias.

Para poder implementarlo, es necesario actualizar, luego de cada referencia, una lista que ordena los bloques por ultimo acceso, indicando cuan reciente fue la última referencia a cada bloque. Si se produce un fallo, se reemplaza aquel bloque que está ultimo en la lista, pero si ocurre un acierto, el bloque accedido debe convertirse en el primero de la lista.

Por ejemplo, considerar una memoria principal de 64 celdas (direcciones de 6 bits), y una memoria caché con 4 líneas y 8 celdas por bloque. La política LRU para una lista de accesos de ejemplo se muestra en la tabla 1

Dirección	N. Bloque	A/F	Lista de bloques
111000	111	Fallo	111
011001	011	Fallo	011,111
001111	001	Fallo	001,011,111
110000	110	Fallo	110,001,011,111
011111	011	Acierto	011,110,001,111
111111	111	Acierto	111,011,110,001
000111	000	Fallo	000,111,011,110
001111	001	Fallo	001,000,111,011

Table 1: Ejemplo LRU

En una caché completamente asociativa la lista es una sola, pero en una caché asociativa por conjuntos, cada conjunto debe mantener su propia lista ordenada. En ambos casos se necesita almacenar información extra para simular cada lista encadenada.

Diversas simulaciones indican que las mejores tazas de acierto se producen aplicando esta política.

El primero en entrar es el primero en salir

La política **FIFO** (*First In-First Out*) hace que, frente a un fallo, se siga una lista circular para elegir la línea donde se alojará el nuevo bloque. De esta manera, se elimina de la memoria cache aquel que fue cargado en primer lugar y los bloques siguientes son removidos en el mismo orden, similarmente a lo que ocurre en una cola de espera en un banco.

Similarmente para el caso LRU, en una memoria totalmente asociativa por conjuntos se debe mantener el registro de una sola lista, y en una asociativa por conjuntos, una por cada conjunto.

Es posible observar que esta política no tiene en cuenta el principio de localidad temporal. Para hacerlo, considerar el caso donde un bloque es requerido repetidamente, en una memoria caché como la del caso LRU. Cada 4 fallos, dicho bloque es reemplazado por otro, sin importar si es muy usado.

La implementación de la lista puede resolverse a través de un bit que se almacena en cada línea (además del tag y de los datos del bloque) que indica cuál es el bloque candidato a ser reemplazado. De esta manera, solo uno de los

bloques tiene un 1, y los restantes deben tener 0. Cuando ocurre un fallo, el bloque que tenía un 1 se reemplaza, se resetea el bit y se invierte el bit (de 0 a 1) de la siguiente línea.

Menos frecuentemente usado

En el método **LFU** (*Least Frequently Used*) el bloque a sustituir será aquel que se acceda **con menos frecuencia**. Será necesario entonces registrar la frecuencia de uso de los diferentes bloques de caché. Esta frecuencia se debe mantener a través de un campo contador almacenado en cada línea, pero esto presenta una limitación en el rango de representación. Para mejorar esto, una posible solucion es ir recalculando la frecuencia cada vez que se realice una operación en caché, dividiendo el número de veces que se ha usado un bloque por el tiempo que hace que entró en caché. La contraparte de este método es que el cálculo mencionado presenta un costo adicional elevado.

Algoritmo Aleatorio

Con esta política, el bloque es elegido al azar. En una memoria asociativa por conjuntos, el azar se calcula entre los bloques que forman el conjunto en el cual se ha producido la falla.

Esta política es contraria al principio de localidad, pues lo desconoce; sin embargo, algunos resultados de simulaciones indican que al utilizar el algoritmo aleatorio se obtienen tazas de acierto superiores a los algoritmos FIFO y LRU.

Posee las ventajas de que por un lado su implementación requiere un mínimo de hardware y por otro lado no es necesario almacenar información alguna para cada bloque.

Fuente

1. *Organización y Arquitectura de Computadoras , William Stallings, Capítulo 4*
2. *http://es.wikipedia.org/wiki/Unidad_de_estado_sólido*