*Computers are like Old Testament gods;*

*lots of rules and no mercy.*

**Listas e Resoluções**

[Lista Hierarquia Aqui](http://cin.ufpe.br/~if674/arquivos/2017.2/Aulas/lista-hierarquia-17-2.pdf)

[Lista E/S Multicore Aqui](http://cin.ufpe.br/~if674/arquivos/2017.2/Aulas/lista-io-multicore.pdf)

[Resolução da Lista Hierarquia Aqui](https://docs.google.com/a/cin.ufpe.br/document/d/13LOl2YeYwnUGB8SyzJYPUTed5Z6fj7GvrJ_pv-uarjs/edit?usp=sharing)

[Resolução da Lista E/S Multicore Aqui](https://docs.google.com/a/cin.ufpe.br/document/d/1Qnb4KED679h1VMCebKu_aGQ0-MXAEGxNI2XySFaMnus/edit?usp=sharing)

**Black Market**

[EE 3 2017.1 [Resolvida]](https://docs.google.com/a/cin.ufpe.br/document/d/1_9RzBVM4OtYVCekRUtSASqWQytTXaMs0xN84ifymLuI/edit?usp=sharing)

[EE2 20XX.2 [Resolvida]](https://docs.google.com/document/d/1j-kDPZ12oSVz7wk2zBCKbtXxyh9-y2Noh61NTlkZ5Ms/edit)

[“Resumo de Alguns Períodos Atrás”](https://drive.google.com/a/cin.ufpe.br/file/d/1CCRpuL75qB9ChYM-yl6o2AAvQOOzr-eo/view?usp=sharing)

**Material Complementar**

[Todos os Mapeamentos de Cache, na Prática[Playlist]](https://www.youtube.com/playlist?list=PLEbnTDJUr_IdI9QZ7bkrhMX2ZpNW0dZUo)

[O que é Memória Virtual? [Vídeo]](https://www.youtube.com/watch?v=2quKyPnUShQ)

[Aula de Memória Virtual da Carnegie Mellon University[Vídeo]](https://www.youtube.com/watch?v=2RhGMpY18zw)

*Hierarquia de Memória*

A memória dos computadores atualmente é dividida de modo a formar uma hierarquia: Memórias mais rápidas e menores são as mais “importantes” e acessadas diretamente pelo processador. Conforme essa distância do processador aumenta, a memória aumenta em tamanho, porém fica mais lenta.

A motivação para isso se deve ao **princípio da localidade**: dados que se encontram próximos temporalmente ou espacialmente tendem a serem usados novamente. Por isso, os dados recorrentes são armazenados a curto prazo nas memórias menores e rápidas, que obtém tais informações das memórias a longo prazo de menor hierarquia.

Vale lembrar: uma hierarquia de memória apenas “conversa” com outras de hierarquia imediatamente acima ou abaixo. Não é possível que a memória com menor hierarquia converse diretamente com o processador, por exemplo.

Outro motivo pelo qual se vale a pena hierarquizar é a relação custo/desempenho dos diferentes tipos de memórias que temos disponíveis:

SRAM (static random access memory) - Extremamente cara, mas absurdamente rápida. Com ela fazemos as **caches.**

DRAM (dynamic random acess memory) - Preço médio, e velocidade razoável. Porém os dados não permanecem por muito tempo, sendo necessária a atualização constante (por isso dynamic). Com ela são feitas as memórias RAM do computador.

DISCO MAGNÉTICO - Extremamente barato, mas terrivelmente lento. Os dados permanecem mesmo quando o computador é desligado, mas se o disco fosse usado diretamente, o processador seria “subestimado”: ele levaria mais tempo esperando o dado chegar do disco do que efetivamente processando.

Por mais rápido que o processador seja, a memória precisa tentar acompanhar essa performance.

Na prática, é o que vemos nos computadores do dia a dia: Uma cache geralmente de 4MB, uma memória RAM de 4GB a 16GB, e os discos magnéticos com 1TB para mais.

***“Memória Ideal:*** *Tempo de acesso de uma SRAM,*

*Capacidade e custo de um disco.”*

**Definições de termos usados na Hierarquia de Memória:**

**Bloco:**

Menor unidade que é copiada da memória principal (RAM) para a cache: Pode ser uma ou várias palavras.

**Hit e Hit Ratio:**

Se o dado já estiver presente na cache quando for necessário acessá-lo, chamamos de **hit.** Podemos medir o **hit ratio** dividindo o número de hits pela quantidade de acessos que realizamos.

**Miss e Miss Ratio:**

Se o dado não estiver presente na cache quando for necessário acessá-lo, chamamos de **miss.** Podemos medir o **miss ratio** dividindo o número de misses pela quantidade de acessos que realizamos.

**Miss Penalty:**

Porém, se o dado está ausente, precisamos **buscá-lo**, e isso **leva tempo**. A isso damos o nome de **miss penalty**, ou seja, é a penalidade que você sofre quando não encontra o dado: Agora terá que buscá-lo da memória e esperar que ele chegue na cache.

**Princípio da Localidade Espacial:**

O programa irá usar em breve instruções e dados que estão próximos (na memória) aos que acabou de usar.

**Princípio da Localidade Temporal:**

O programa irá usar em breve instruções e dados que usou recentemente.

**Cache:** É uma memória de acesso muito rápido, mas de tamanho reduzido. Serve para manter os dados mais recentes sendo usados pelo processador, e possui um controlador próprio que determina o que será transferido da memória principal para a cache.

**Vantagens:**

Tira proveito do princípio da localidade espacial e temporal: dados usados recentemente e dados próximos aos usados recentemente provavelmente serão usados de novo.

Otimiza as diferentes tecnologias que relacionam custo e tempo de acesso: memórias mais rápidas e menores são mais caras, e memórias grandes porém lentas são baratas. O que se deseja atingir com a hierarquia é o custo e tamanho de uma memória grande como o HD, mas ter o tempo de acesso de uma SRAM (memória do tipo mais caro)

Dado a essa otimização, o desempenho da CPU aumenta: as memórias não se desenvolveram no mesmo passo dos processadores, o que faz com que, sem um planejamento do sistema de memória, a capacidade do processador tenha um gargalo, sendo impedido de operar com máxima eficiência.

**Desvantagens:**

Maior complexidade na implementação e custo elevado.

*Layouts de Cache*

Ao elaborar as memórias cache, era necessário estabelecer um modo de identificar se uma informação estava na cache e, caso ela estivesse, como encontrá-la. Para isso, alguns métodos foram desenvolvidos:

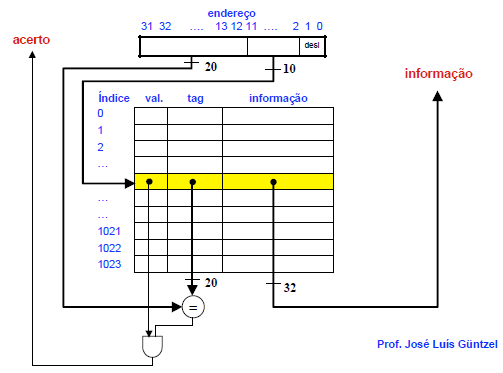
**Mapeamento Direto**

Para cada palavra na cache, atribuir um endereço com base no endereço da palavra na memória principal. A maioria das caches que usa mapeamento direto o faz usando o seguinte processo:

*(Endereço ao byte/bytes por bloco) módulo (Número de blocos da cache)*

Como mais de uma entrada da cache pode armazenar o conteúdo de mais de um endereço da memória principal, é utilizado uma *tag* em conjunto com o endereço do mapeamento, de modo a compor o endereço completo, com relação à memória principal. É utilizado também um *bit de validade* para identificar se a informação contida naquele slot da cache é válido.

Com isso, um exemplo de cache de mapeamento direto é dado a seguir:



Para calcular o **tamanho de uma cache de mapeamento direto** é necessário conhecer o tamanho do bloco, tamanho da tag e tamanho do campo de validade:

2n x (tamanho do bloco + tamanho do tag + tamanho do campo de validade)

Já o **layout do endereço de uma cache com mapeamento direto** contém a *tag,* o *índice (mostrado logo acima como calcular)*, o *block offset (indica a word referenciada dentro do bloco)* e o *byte offset (indica o byte dentro da word)*.

Ao calcular a quantidade de bits necessária para endereçar todas as words dentro de um bloco, calcula-se a quantidade de bits de um block offset. Ao calcular a quantidade de bits necessárias para endereçar todos os bytes dentro de uma word, calcula-se a quantidade de bits de um byte offset.

**Associativa por Conjunto (Bloco Associativo)**

Existe um número fixo de posições na cache (no mínimo duas) nas quais cada bloco pode ser colocado. Uma cache n-associative é uma cache associativa por conjunto com n posições possíveis para guardar um bloco. Tal cache é composta por um certo número de conjuntos, cada conjunto com n blocos. Cada bloco da memória principal é mapeado para um dos conjuntos da cache, determinado pelo campo de índice do endereço (sendo que o bloco pode ser colocado em qualquer dos elementos desse conjunto).

O conjunto que contém o bloco de memória é dado por

*(Número do bloco) módulo (Número de conjuntos da cache)*

O aumento do grau de associatividade resulta, em geral, na redução da taxa de faltas. A desvantagem é o aumento do tempo de tratamento do acerto.

**Totalmente Associativa**

Nesse mapeamento, um bloco da memória principal pode ser colocado em qualquer posição da cache. Um bloco de memória pode ser associado a qualquer “entrada” da cache. É, então, necessário pesquisar todas as entradas da cache, a fim de localizar um determinado bloco (pois ele pode estar em qualquer uma das entradas). Para reduzir o tempo de pesquisa, ela é feita em paralelo, usando um comparador para cada entrada da cache.

O custo do hardware é alto: tal esquema só é atrativo para caches pequenas.

A **escolha** entre um **mapeamento direto**, **associativo por conjunto** ou **totalmente associativo** irá depender do custo de uma falta versus o custo de implementação da associatividade, tanto em termos de tempo quanto em termos de hardware.

*Misses*

**Tipos de Misses**

**Compulsórios:** Impossíveis de tratar. São misses que obrigatoriamente acontecerão, pois são os que ocorrem quando a cache está inicialmente vazio. São também chamado de misses de “partida a frio”.

**Conflito:** Misses que acontecem devido à disputa de dados para ficar no mesmo slot de cache. Não acontecem numa cache completamente associativa.

**Capacidade:** Misses que acontecem porque, mesmo que não houvesse nenhum miss de conflito, seria fisicamente impossível colocar o dado sem retirar outro, pois a cache encontra-se cheia.

**Como Lidar com Misses**

Se a cache informar uma falta o processador deve ser parado (congelando o conteúdo de todos os registradores), um controlador separado ajuda no tratamento das faltas geradas no acesso à cache, comandando a busca da informação (bloco) na memória principal ou na cache de próximo nível. Uma vez que o dado tenha sido obtido, a execução é reiniciada no ciclo que gerou a falta no acesso a cache.

**Misses de conflito** podem ser tratados aumentando a associatividade da cache, porém isso faz com que ela fique significativamente mais cara: são necessários mais comparadores, e o hardware necessário torna-se mais complexo.

Já misses de capacidade podem ser tratados simplesmente aumentando o tamanho da cache.

**Penalidade por Falta**

À medida que o tamanho do bloco aumenta, aumenta o custo de uma falta. A **penalidade por falta** é determinada pelo tempo necessário à busca de um bloco no nível imediatamente inferior na hierarquia, carregando-a na cache.

O tempo de busca é dividido em duas partes: a **latência** para a busca da primeira palavra e o **tempo de transferência** do resto do bloco.

*Medidas do desempenho da cache*

Tempo de processador = (Ciclos de relógio gastos na execução normal do programa + ciclos de relógio gastos à espera do acesso ao sistema de memória) x tempo de clock

Onde,

Ciclos de relógio gastos à espera do acesso ao sistema de memória = ciclos de clock para tratar faltas de leitura da cache + ciclos de relógio para tratar faltas de escrita na cache

Ciclos de clock para tratar faltas na cache = (Acessos à memória por programa) x (faltas por instrução) x Penalidade por faltas

*Melhorias da Cache*

**Políticas de Escrita**

São duas: **write-back** e **write-through.**

Ambas as políticas tratam de um problema existente ao se usar memórias cache: Devido à hierarquia de memória, o processador irá modificar dados na cache, por ser mais rápida e de fácil acesso. No entanto, os valores na memória principal agora estão desatualizados! Podemos resolver isso mantendo ambas sempre atualizadas: ou seja, sempre que a cache for modificada, a memória principal também será atualizada, em sincronia. A essa técnica damos o nome de **write through.** Esse sistema consegue, de forma simples de se implementar, manter os dados sempre consistentes. O problema disso é que é muito lento fazer isso toda vez que uma mudança for feita na memória. Isso arruinaria todo o propósito de manter uma cache para se aproveitar de sua velocidade. Uma solução paliativa para essa diminuição do desempenho ao utilizar o **write through** é utilizar um buffer de escrita, no qual os dados serão armazenados enquanto aguardam para serem escritos na memória. Após escrever o dado na cache e no buffer de escrita, o processador pode continuar a execução das instruções. Se o buffer de escrita estiver cheio quando o processador tiver que executar uma instrução de escrita, o processador precisa parar, até que haja posição disponível no buffer.

A outra abordagem, chamada **write back**, mantém um bit extra para cada slot de cache, chamado “**dirty bit**”. Esse bit é responsável por indicar se o conteúdo está “sujo”, ou seja, se ele sofreu modificações. Dessa forma, a memória principal será atualizada, mas somente quando o slot sofrer uma substituição e o dirty bit estiver em 1. Dessa forma, todas as mudanças serão feitas de uma só vez, e somente quando há a troca de slots. A desvantagem de um sistema desses é o aumento da complexidade para sua implementação, por exemplo, num computador multiprocessado, manter a consistência de cache necessita de cuidados a mais.

Para melhorar a política de **write back,** podemos usar um write-buffer. As mudanças vão sendo salvas neste buffer, que monitora o barramento. Quando o barramento estiver livre, as mudanças são escritas na memória. Isso diminui a competição no uso do barramento e possíveis gargalos na performance.

**Melhorando a Performance da Cache**

* **Reinício Precoce (Early Restart):** quando ocorre uma falta, o processamento é congelado e o bloco é substituído, com essa técnica, a execução é retomada quando a word solicitada é retornada, ao invés de esperar todo o bloco ser escrito.
* **Word Crítica Primeiro:** a word requisitada é transferida primeiro para a memória cache.
* **Cache multinível:** nessa abordagem, são utilizados vários níveis de cache, onde quanto mais afastada do processador, maior o armazenamento e o tempo de acesso, porém menor o custo é menor. Assim, cada nível pode focar em melhorar um aspecto da performance da cache. Uma estrutura de dois níveis de cache, por exemplo, permite que a primária se concentre em minimizar o tempo de acerto, porém, como a quantidade de misses seria a mesma, ela pode fazer uso da cache secundária, que estará focada em maximizar sua capacidade e reduzir a penalidade de uma falha direta na memória principal.
* **Aumento no Tamanho do Bloco:** É utilizado para tirar proveito da localidade espacial, visto que, ao ocorrer uma falta, é provável que palavras adjacentes serão necessárias em breve. Pode resultar em um miss rate menor, devido à localidade espacial, porém ao custo de aumentar a penalidade (blocos maiores levam mais tempo para serem transferidos). No entanto, após um certo limite, blocos muito grandes se tornam um empecilho: devido ao tamanho, ocupam espaço demais e as trocas constantes deles na cache não valem a pena.
* **Aumento de banda passante da memória principal para a cache:** esta redução no custo da penalidade permite o uso de blocos maiores, a um custo próximo daquele obtido com blocos pequenos.

*Memória Virtual*

A técnica de memória virtual realiza a tradução do espaço de endereçamento de um programa para seus endereços reais. Permite que o tamanho de um único programa exceda a quantidade total de memória real disponível para sua execução. Essa técnica gerencia automaticamente os dois níveis de hierarquia: memória principal (física) e memória secundária.

**Definições**

**Página:** bloco de tamanho fixo.

**Falta de página:** é uma falta no acesso à memória virtual.

O processador sempre gera um **endereço virtual**, que é traduzido para um endereço real por meio da MMU (memory management unit), que é um sistema HW+SW. O **endereço real** (também chamado de físico) é, então, usado para acessar a memória.

**Memória Virtual e Relocação de código**

A técnica da memória virtual simplifica a carga dos programas para execução, a partir da realocação: ela mapeia os endereços virtuais usados por um determinado programa em endereços físicos (antes de tais endereços serem usados para acessar a memória). Essa técnica de realocação permite que um programa seja carregado em qualquer posição da memória principal. Todos os sistemas de memória virtual realocam os programas por meio de blocos de tamanho fixo: as páginas.

A ocorrência de uma falta de página consome **milhões de ciclos de relógio**, este tempo é dominado pelo tempo para obter o primeiro endereço de uma página de tamanho típico. Para amortizar o tempo de acesso muito grande, as páginas devem ser grandes o suficiente. Uma técnica de **colocação totalmente associativa** tende a proporcionar uma baixa taxa de falta de página.

Os sistemas de memória virtual usam **write-back.**

**Tabela de Página e TLB**

A redução das faltas de página é o principal objetivo no projeto de um sistema de memória virtual, esta redução pode ser obtida através do uso de técnicas otimizadas para a colocação das páginas na memória principal.

* **Tabela de páginas**

Permite a localização das páginas indexando totalmente a memória principal e sendo lá armazenada. É indexada com o número da página extraído do endereço virtual e contém o número da página física correspondente;

Cada programa possui sua própria tabela de páginas;

**Registrador da tabela de páginas:** em HW, aponta para a posição inicial da tabela de páginas na memória principal

* **Falta de página**

A tabela de páginas possui um **bit de residência** que quando igual a 0indica falta de página;

Durante as faltas de página, o sistema operacional assume o controle, por meio do mecanismo de exceção.

O endereço virtual, por si só, não informa em que posição do HD está a página que gerou a falta de página.

* **TLB**

Cada acesso à memória exige, na verdade, dois acessos à memória principal: um para obter o endereço físico (consulta à tabela de páginas) e outro para buscar a informação.

Os processadores atuais possuem uma cache especial (muitas vezes, on-chip) que armazena as traduções de endereço mais recentes: **translation-lookaside buffer** - **TLB**.

Naturalmente, a TLB possui menos entradas do que o número de páginas da memória principal.

Dada uma entrada da TLB, o rótulo guarda uma parte do número da página virtual e o campo de informação guarda o número do endereço físico correspondente. Em uma referência, a tabela de páginas pode nem ser acessada. Logo, cada entrada da TLB precisa ter também: o bit de residência e um bit de modificação, para o caso de escrita.

Caso ocorra uma falta no acesso à TLB, é preciso determinar se realmente foi falta no acesso à TLB ou se foi falta devida à falta de página. Se a página estiver na memória principal (falta no acesso à TLB), o processador trata a falta colocando na TLB as informações necessárias para realizar a tradução (as informações são pegas na memória principal) e, assim, o processador tenta novamente acessar a página a partir da TLB. Se a página não estiver na memória principal (falta na TLB indica falta de página) o processador aciona o SO através através de uma exceção.

É importante salientar que o nº de entradas na TLB < nº de páginas na memória principal. Faltas na TLB são muito mais frequentes que faltas de página.

**Modos de Tradução de Endereço**

**Paginação:** esse modo de tradução consiste em dividir a memória em blocos que são chamados de páginas. A alocação de memória é requisitada por essas páginas, sendo a unidade de transferência entre a memória e o disco. É necessário uma tabela de páginas para saber onde qual endereço virtual corresponde na memória física.

Uma vantagem dessa técnica é a impossibilidade de acontecer fragmentação externa, já que as páginas de um processo não precisam ficar contínuas na memória. A desvantagem é a existência da fragmentação interna, ou seja, desperdício de espaço por consequência do tamanho de páginas ser fixo.

**Segmentação:** já nesse modo, os processos são divididos em segmentos de tamanhos variados. A alocação é feita por uma tabela de segmentos, que guarda o início do segmento e o tamanho.

É o contrário da paginação: pode existir fragmentação externa mas não pode existir fragmentação interna. Outra vantagem é a facilidade de proteção, já que cada processo tem seu segmento e tamanho determinados.

**Segmentação Paginada:** para contornar as desvantagens anteriores, é possível dividir os segmentos em páginas, esta também permite que segmentos maiores que a memória possam ser carregados. Uma desvantagem é quantidade de tabelas, aumentando o acesso à memória.

*Entrada e Saída*

**RAID**

**Barramento e Comunicação**

**Barramento Síncrono e Assíncrono**

**Síncrono:**

Barramentos síncronos necessitam pouca lógica, podendo ser facilmente implementados com FSM, mas é preciso que todos dispositivos operem na mesma frequência de clock, além disso, barramentos não podem ser tão longo se a taxa de transferência for alta.

**Assíncrono:**

Um barramento assíncrono não é guiado pelo clock. Dessa forma, ele pode acomodar uma diferente gama de dispositivos com diferentes velocidades, e o barramento pode ser ampliado sem o risco de ficar “fora de sincronia”. Para coordenar a transferência de dados, um barramento assíncrono utiliza um **handshaking protocol**, ou seja, são usados sinais de “requerimento” e de “pronto”. A transmissão do dado ocorre somente quando todas as partes envolvidas estiverem prontas para enviar/receber.

**Controlando Acesso ao Barramento**

O CPU não pode ser usado diretamente para controlar o acesso ao barramento. Muitas interrupções diminuem a eficiência do processador. Podemos ter um **árbitro** centralizado, que decidirá que dispositivo usará o bus, inclusive um **árbitro híbrido**, centralizado e distribuído, onde existe prioridades. Nessa última abordagem, os dispositivos são ligados em série, os mais próximos do árbitro possuem maior prioridade.

O árbitro é o **bus master**, que decide agora quem será o próximo a usar o barramento, e pode iniciar e controlar requisições de uso ao barramento.

**Tipos de Comunicação entre I/O e o Processador:**

**Polling:** É o jeito mais simples de um dispositivo de E/S se comunicar com o processador. O dispositivo coloca a informação num registrador de status, que o processador está periodicamente checando para ver se há alguma alteração. A desvantagem disso é que o processador é muito mais rápido: ele pode checar o registrador várias vezes, sendo que o dispositivo pode não ter feito alteração alguma (como por exemplo, o mouse não ter se mexido nas últimas 1000 checagens).

O uso ideal para polling é quando as taxas de alteração dos dispositivos de E/S são **predeterminadas.** Assim, o overhead é mais previsível. É geralmente usado em sistemas de tempo real.

**Interrupt Driven:** É um esquema que faz uso de interrupções para dizer quando o dispositivo de E/S precisa da atenção do processador. Mas isso não quer dizer que ele seja síncrono, dependente de instrução, ou que pode evitar que uma instrução seja executada. A checagem se existe algum dispositivo de E/S pendente de ser tratado é feito antes de iniciar uma nova instrução. Feita a comunicação, geralmente o restante do processamento é gerenciado pelo sistema operacional. (Como por exemplo, uma tela pressionada do teclado)

Interrupt Driven Communication é usada por quase todos os sistemas em pelo menos alguns de seus dispositivos de E/S.

**DMA:** Significa Direct Memory Access, e é implementado com um controlador especial. Esse controlador assume controle do barramento (bus master), o que significa que ele pode iniciar requisições. O DMA é usado porque o overhead usando um sistema Interrupt Driven para transferir dados do disco rígido é absurdo. Por isso, o DMA é usado como um assistente - ele livra o processador de ficar constantemente checando se o dado foi transferido, realiza a transferência de forma autônoma, sem “incomodar” o processador para que ele possa continuar realizando outras tarefas, e só realiza uma interrupção novamente quando a transferência for concluída, para que o processador ou ‘pergunte’ ao DMA ou cheque na memória se a transferência foi bem sucedida.

Processadores DMA geralmente são processadores de propósito único, e são implementados para realizar essa interface entre o disco rígido e o processador, quando uma transferência se faz necessária.

**Proteção de Acesso pelo S.O.**

O SO precisa gerir as permissões de cada programa-usuário, um programa qualquer não pode escrever no disco sem permissão concedida. Técnicas como "Read Write Execute" podem ser utilizadas para garantir que programas só leiam, escrevam e no que lhe for permitido.

O SO faz isso impedindo que o programa do usuário se comunique com os dispositivos de E/S diretamente. Existem formas de realizar esse intermédio:

* Dar comandos aos dispositivos de E/S (read, write, seek, etc)
* O dispositivo de E/S notificar o SO quando completar uma operação ou encontrar um erro.
* Os dados só poderão ser transferidos da memória para o dispositivo de E/S. Nunca se pode ler ou escrever diretamente do disco, por exemplo.

*Multi-Core*

**Modelos de Memória de Multi-Cores**

Num multi core, é utilizado um mesmo espaço de endereçamento de memória para todos os processadores. Eles se comunicam através de variáveis compartilhadas, e todos os processadores podem acessar qualquer lugar na memória por meio de loads e stores.

Os processadores multi-cores podem ser classificados em dois tipos, de acordo com os modelos de memória: **Memória Centralizada** (SMP ou Symmetric Multiprocessors, também chamado de UMA - Uniform Memory Access) e **Memória Distribuída** (NUMA - Non Uniform Memory Access).

Numa abordagem **UMA**, os acessos à memória principal levam o mesmo período, independente do processador que requisita o acesso e de qual word é requisitada.

Já na abordagem de **NUMA**, alguns acessos à memória são mais rápidos que outros, dependendo de que processador requisita que word.

Também existem os processadores que não fazem uso de memória compartilhada: eles possuem apenas seções de memória **privada**, e se comunicam através de troca de mensagens.

**Modelos de Comunicação em Processos Paralelos**

Temos o **message-passing,** onde os processadores trocam mensagens por meios de uma rede local. Para isso, é necessário que o sistema tenha rotinas para comunicação. O modelo de memória mais adequado nesse caso, é o de memória privada, muito utilizado em clusters.

Também existem os processadores baseados em **memória compartilhada**, que se comunicam por meio de variáveis compartilhadas usando loads e stores. Nesse caso, a abordagem mais utilizada seria a NUMA, que distribui fisicamente a memória entre os processadores. A abordagem UMA também é viável, mas pouco utilizada.

**Resolvendo Coerência de Cache**

A coerência de cache diz respeito a **consistência** **no valor** dos dados entre as versões nas caches de vários processadores. Como cache é atualizada constantemente, é importante que a seus dados estejam coerentes com a memória principal, além disso, em sistemas multiprocessados, cada CPU pode ter uma cache própria, mas com dados compartilhados, é importantes que os dados sejam lidos e escritos corretamente. Ao atualizar seus dados, o processador não deveria se importar se o dado é compartilhado ou não causando várias cópias discrepantes de um mesmo dado.

O problema de coerência de cache surge quando a memória é compartilhada por vários processadores. Uma cache privada não sofre deste problema pois somente um processador pode acessá-la.

Esse problema pode ser resolvido nas caches compartilhadas através da implementação de **protocolos** pelo controlador da cache ou pelo controlador de memória que irão cuidar de **rastrear o estado de compartilhamento dos dados.**

Dois dos protocolos mais comuns são: **snooping** ou **baseada em diretório**.

No protocolo de **snooping**, cada cache que tem uma cópia do dado também tem uma cópia do estado de compartilhamento. Todas as caches são acessíveis através de um mesmo barramento, e os controladores de cache ficam monitorando este barramento (snooping) para saber se alguma cópia que eles detém está sendo requisitada ou não por outras caches.

O protocolo de snooping usa de duas técnicas para manter a consistência dos dados: **write update** e **write invalidate.** No primeiro, após acontecer uma escrita em uma das caches, todas as caches que possuírem uma cópia desse dado atualizam seus valores. No caso do invalidate, sempre que ocorre uma escrita numa variável compartilhada, as caches que possuem uma cópia tratam de invalidá-las. Como o protocolo write update causa maior uso do barramento para manter as variáveis atualizadas, o mais usado é o write invalidate.

No protocolo **baseado em diretório**, o status de compartilhamento de um bloco da memória principal é armazenado em um local centralizado (diretório). Toda mudança feita deve ser comunicada ao diretório, que cuida de comunicar a quem for necessário (quem tiver cópias dos dados) as alterações feitas. Um único diretório mantém o estado de cada bloco na memória principal. As informações no diretório podem incluir quais caches possuem cópias do bloco, se ele é sujo etc.

Se você chegou aqui, parabéns!



nAruTO-UpsIDE-dOWn

Ta-eM-ChOK? kk