Simulador de Hierarquia de Memória em Multicore

IMD0121 - Arquitetura de Computadores

Ranna Raabe Fernandes da Costa
Victor Ângelo Graça Morais

1 Introdução

A hierarquia de memória se refere a uma classificação das memórias divida em duas dimensões: capacidade de armazenamento e velocidade de acesso. Essa hierarquia geralmente é representada por uma pirâmide, onde no topo encontram-se as memórias menores em tamanho, mais rápidas e que possuem menor capacidade, enquanto na base da pirâmide estão as memórias maiores em tamanho, mais lentas e com maior capacidade. Consequentemente, o topo possui as memórias mais caras e a base possui as memórias mais baratas em valor.

A importância da hierarquia se dá devido ao fato de que, ter uma única memória responsável por tudo, é custoso em dinheiro e em tempo de processamento, dificultando o desempenho da máquina.

2 Abordagem

O Simulador de hierarquia de memória em multicore possui dois níveis de cache (L1 e L2) e a memória principal. É utilizado para monitoramento dos dados na memória principal. Cada core do computador possui uma cache L1, e uma cache L2 é compartilhada entre os cores do processador. A memória é compartilhada por todos os cores, como descrito na figura abaixo.

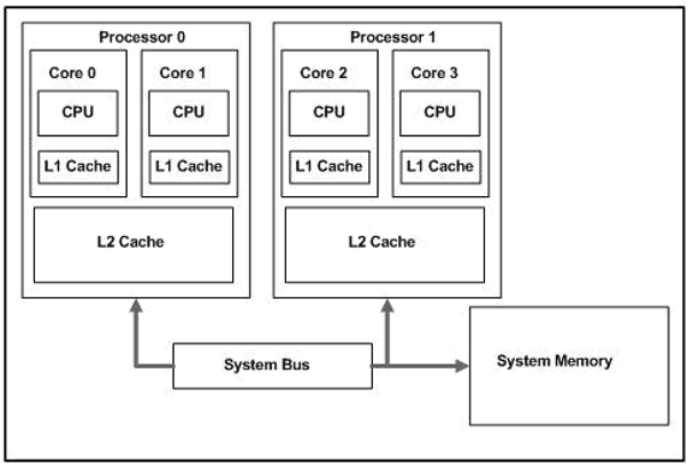

Figura 1: Modelo de Hierarquia de Memória em Multicore

O simulador foi implementado em Java, utilizando conceitos e paradigmas da linguagem. A implementação foi feita baseada na figura que representa o modelo de hierarquia de memória em multicore. O simulador cria as as caches e a memória principal automaticamente, cada processador possui **4 cores**, **2 CPU's** e **600 de memória**. O simulador lê um arquivo e carrega os dados na memória principal.

Em seguida, o usuário informa o endereço da memória principal para realizar a leitura e informa também o core que utilizará aquele dado, e o sistema carrega o dado na cache L2 e na cache L1 do core. Caso o core modifique esse valor, o simulador atualiza todos os níveis da hierarquia imediatamente, utilizando a técnica write-through. O código de desenvolvimento do simulador pode ser encontrado no link: `https://github.com/rannaraabe/arq-simulator`.

2.1 Diagrama de classes

O diagrama de classes do simulador foi definido da seguinte forma:

Figura 2: Diagrama de Classes

2.2 Core

A classe Core possui um atributo cache L1, como mostra o modelo de hierarquia de memória, e um atributo inteiro que representa o tamanho do bloco da cache L1. Além do mais, a classe Core possui métodos para resgatar (*getCacheL1()*) e setar valores (*setCache()*) da cache L1 de cada Core, e também o método *setBlkSize()* que seta o tamanho do bloco da cache L1.

Além destes, possui também o método *l1HasMemory()* que verifica se o dado está na cache L1. O método *loadToCacheL1()* que recebe uma posição na memória e carrega o bloco na cache L1 através do mapeamento direto.

O outro método *loadToCacheL1()* que recebe uma posição na memória e a memória, imprime no console o dado que está naquele índice passado por parâmetro daquela memória. O método *execute()* realiza a mudança de dados em um índice da memória, e mostra ao usuário a mudança que foi feita.

Por fim, o método *printL1()*, o método que imprime no console para o usuário a cache L1 (que por padrão definido possui tamanho 30) com seus dados escritos.

2.3 CPU

A classe CPU possui como atributos um vetor que representa os Cores, um vetor que representa a cache L2 (como mostra o modelo de hierarquia de memória, a cache L2 é compartilhada entre as CPU's, diferente da cache L1) e um atributo inteiro que representa o tamanho do bloco da cache L2.

Possui métodos *setCpu()* que inicializa a CPU com número de Cores, tamanho da cache L1 e tamanho da cache L2, método *getCores()* que recupera os Cores, método *input()* que carrega a cache L1 em um core passado por parâmetro.

A classe também possui método *l2HasMem()* que verifica se o dado está na cache L2 ou não. O método *memIsLoad()* que verifica primeiramente se o dado foi carregado na cache L1, caso tenha sido carregado na cache L1 ele confere se também está na cache L2 e informa ao usuário, caso não esteja na L2 ele carrega o dado lá e informa ao usuário; analogamente, o simulador verifica se foi carregado na cache L2, e informa ao usuário caso o dado esteja na L2 e não esteja na L1 e carrega o dado na L1. Caso o dado não esteja na cache L1 e também não esteja na cache L2, o sistema carrega o dado nas duas caches.

O método *needToRemove()* verifica se existe algum índice de memória que precisa ser removido na cache L1. O método *input()* que recebe índice da Memória, índice do Core, memória e valor a ser escrito, faz chamada ao método *execute()* (presente na classe Core) que realiza a mudança de dados em um índice da memória, colocando o valor a ser escrito que foi passado por parâmetro do método *input()*, e mostra ao usuário a mudança que foi feita.

Por fim, o método *printL2()*, é o método que imprime no console para o usuário a cache L2 (que por padrão definido possui 60 espaços) com seus dados escritos.

2.4 Memory

A classe Memory é a classe mais simples do sistema, possui apenas dois atributos: um vetor de inteiros representando a memória e um inteiro que representa os índices da memória. O construtor parametrizado do sistema, inicializa a memória com um tamanho *size* passado.

Além disso, possui uma função *add()* que recebe uma String que representa o dado a ser inserido na memória, caso a memória não esteja cheia o sistema adiciona o dado no índice referido, caso a memória esteja cheia o sistema retorna uma mensagem informando que não é possiível adicionar o dado pois a memória está cheia. E por fim, possui uma função que retorna a memória, o método *getMemory()*. E o método *printMemory()* que imprime a disposição dos dados da memória no console.

2.5 PC

A classe PC é a principal responsável pelos métodos de leitura e escrita da memória. Possui um vetor de CPUS, uma memória, um inteiro que representa um índice na memória, um inteiro que representa um índice na CPU e um inteiro que representa um índice no Core.

O construtor parametrizado da classe inicializa um PC com um tamanho de memória, números de CPU, números de Cores e arquivo que será lido pelo simulador. O método *toMem()* da classe, realiza a leitura dos dados que estão no arquivo de texto passado pelo usuário. O método *choose()*

possui apenas prints para solicitar ao usuário o índice da memória, a CPU e o core desejado para leitura ou escrita, e Scanner para realizar a leitura destes dados que serão informados pelo console.

Em seguida, os métodos de leitura e escrita, *read()* e *write()* que estão nesta classe realizam as seguintes operações, respectivamente: verifica se o endereço da memória está carregado na cache L2 ou L1, caso não esteja carregado, carrega-o; faz a leitura do valor que será escrito na memória, verifica se o endereço da memória está carregado, caso não esteja carregado, carrega-o e depois escreve na memória, atualizando em todos os níveis da hierarquia.

Por fim, a classe possui o método *printMemory()* que recebe um booleano. Este método é o responsável para imprimir no console o status da memória, ou seja, o que está escrito em cada posição da memória. Caso o booleano passado na função seja true, o método imprime também o status da CPU e o status do Core.

2.6 Descrição do Sistema

O simulador funciona da seguinte forma: ao executar, é solicitado um arquivo de texto com dados que serão carregados na memória inicialmente, em seguida o usuário escolhe entre fazer leitura ou escrita na memória, após fazer sua escolha será solicitado ao usuário o endereço da memória (como a memória está sendo representada por um vetor, ele informa um índice entre 0 e 599), o core que utilizará aquele dado (entre 0 e 2) e a CPU correspondente (entre 0 e 1).

Primeiro, o sistema solicita o arquivo de texto (./arq-simulator/txt/) que possui os dados a serem carregados na memória. Em seguida, o usuário informa qual opção deseja executar: leitura ou escrita. Caso deseje encerrar o programa, basta digitar 'e'.

```
Main (2) [Java Application] /usr/lib/jvm/jdk-oracle-8u231-linux-x64/jdk1.8.0_231/jre/bin/java (18 de nov de 2019 19:45:32)
Reading file from txt folder... Enter file name:
test


Choose one:
1 - read
2 - write
>>> Type e to exit
```

Figura 3: Solicitação do arquivo e opções de escrita ou leitura dadas ao usuário

Caso sua opção seja a leitura, ele informa o endereço da memória e o core que utilizará aquele dado, e o simulador retorna o status da memória com os dados presentes nela. A primeira linha contém os índices da memória, e a segunda contém os dados que estão escritos nestes índices da memória.

```
Main (2) [Java Application] /usr/lib/jvm/jdk-oracle-8u231-linux-x64/jdk1.8.0_231/jre/bin/java (18 de nov de 2019 19:45:32)
Reading file from txt folder... Enter file name:
test


Choose one:
1 - read
2 - write
>>> Type e to exit
1
Select memory to be read[0, 599]:
206
Select a CPU[0, 1]:
0
Select a core[0, 3]:
2
>>> Memory state:
0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38|39|40|41|42|43|44|45|46|47|48|49|50|51|52|53|54|55|56|57|58|
49|50|51|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|

>>> Memory Address is not in Cache l2 or Cache l1
>>> Loading Memory Address to Cache l2 and l1...
Memory Address: 206
Data: 0
```

Figura 4: Leitura do dado na memória

Semelhante à escrita, o usuário informa o endereço da memória e o core que utilizará aquele dado, e o simulador retorna o status da memória com os dados presentes nela. E em seguida o valor que ele deseja escrever na memória. E o sistema informa se o dado está nas caches ou não.

```
Choose one:
1 - read
2 - write
>>> Type e to exit
2
Select memory to be read[0, 599]:
35
Select a CPU[0, 1]:
1
Select a core[0, 3]:
3
>>> Memory state:
0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38|39|40|41|42|43|44|45|46|47|48|49|50|51|52|53|54|55|56|57|58|59|60|
49|50|51|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|

Enter new value:
267
>>> Memory Address is not in Cache l2 or Cache l1
```

Figura 5: Escrita do dado na memória

3 Conclusão

O desenvolvimento do simulador de hierarquia em memória multicore foi realizado como previsto e com êxito, de forma a atender todas as necessidades propostas.

É importante destacar que algumas informações foram consideradas ao implementar o sistema, foram elas: os dados (escritos e lidos) são inteiros e exploram o princípio da localidade, o tamanho da linha da cache L2 é múltiplo da linha da cache L1, o bloco da memória múltiplo do tamanho da linha da cache L2 e coerência de cache.