

Programação Concorrente em Java

Aula 09

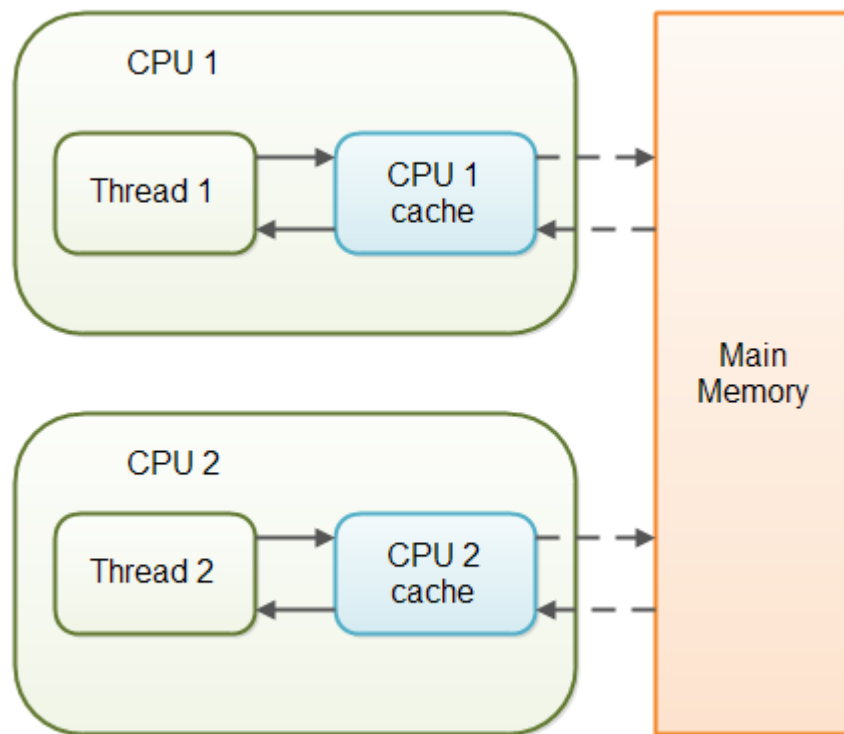
A Palavra Reservada Volatile

Introdução

- A palavra reservada **volatile** para marcar uma variável como “reservada na memória principal”.
- Isso quer dizer que qualquer operação em uma variável volátil será feita na memória principal. O mesmo para a operação de escrita.

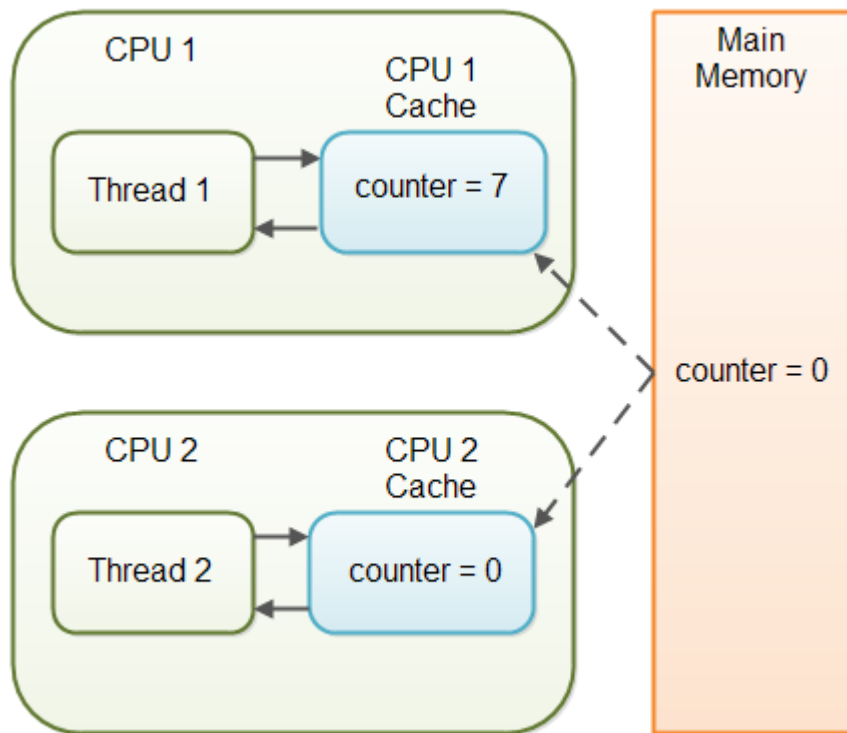
Problemas de visibilidade

- A palavra **volatile** garante a visibilidade de mudanças entre as threads.
- Em um ambiente multithread, variáveis são copiadas para a cache de CPUs, poder motivos de otimização.



Problemas de visibilidade

```
public class SharedObject {  
    public int counter = 0;  
}
```



Garantia da visibilidade

- A palavra **volatile** foi criada com o intuito de sanar problemas de visibilidade.
- Ao declarar **counter** como volatile, todas as operações de escrita são diretamente atualizadas na memória principal, evitando que outra thread leia um valor desatualizado.

```
public class SharedObject {  
    public volatile int counter = 0;  
}
```

Garantia da visibilidade

- Declarando a variável **volatile**, garante sua visibilidade para outras threads.
- No cenário acima, se uma thread escreve e outra lê, o **volatile** garante que a leitura será feita diretamente da memória, sendo assim, o valor atualizado.
- Entretanto, se ambas as threads **escrevem** na variável, apenas a declaração **volatile** não será suficiente.

Garantia de visibilidade total

- Na verdade, a garantia de visibilidade da variável **volatile** vai além da variável. Por exemplo:

```
public class MyClass {  
    private int years;  
    private int months;  
    private volatile int days;  
    public void update(int years, int months, int days){  
        this.years = years;  
        this.months = months;  
        this.days = days;  
    }  
}
```

Garantia de visibilidade total

- O método `update` escreve em 3 variáveis, dentre as quais uma é **volatile**.
- A garantia de **visibilidade total** diz que quando um valor é escrito em **days**, então todas as variáveis visíveis a thread são também escritas na memória.
- Ou seja, caso **days** seja atualizado, os valores de **years** e **months** são escritos na memória principal.

Garantia de visibilidade total


```
public class MyClass {  
    private int years;  
    private int months;  
    private volatile int days;  
    public int totalDays() {  
        int total = this.days;  
        total += months * 30;  
        total += years * 365;  
        return total;  
    }  
    public void update(int years, int months, int days){  
        this.years = years;  
        this.months = months;  
        this.days = days;  
    }  
}
```

- O valor **days** é lido no início de **totalDays()** e armazenado em **total**.
- Os valores de **months** e **years** também são lidos da memória principal.
- Sendo, é garantido ver seus últimos valores.

Reordenação

- A JVM é autorizada a reordenar instruções para otimizar o tempo de execução, contanto que a semântica não seja alterada. Por exemplo:

```
int a = 1;  
int b = 2;  
a++;  
b++;
```



```
int a = 1;  
a++;  
int b = 2;  
b++;
```

- Entretanto, quando uma das variáveis é **volatile**, a reordenação se torna um desafio.

Reordenação

```
public class MyClass {  
    private int years;  
    private int months;  
    private volatile int days;  
    public void update(int years, int months, int days){  
        this.years = years;  
        this.months = months;  
        this.days = days;  
    }  
}
```

 Qual é o problema dessa reordenação?

```
public void update(int years, int months, int days){  
    this.days = days;  
    this.months = months;  
    this.years = years;  
}
```

Garantia happens-before

- Para garantir a reordenação das instruções, a palavra **volatile** adota a abordagem **happens-before**:
 - Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable. The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable. Notice that it is still possible for e.g. reads / writes of other variables located after a write to a volatile to be reordered to occur before that write to the volatile. Just not the other way around. From after to before is allowed, but from before to after is not allowed.
 - Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable. Notice that it is possible for reads of other variables that occur before the read of a volatile variable can be reordered to occur after the read of the volatile. Just not the other way around. From before to after is allowed, but from after to before is not allowed.

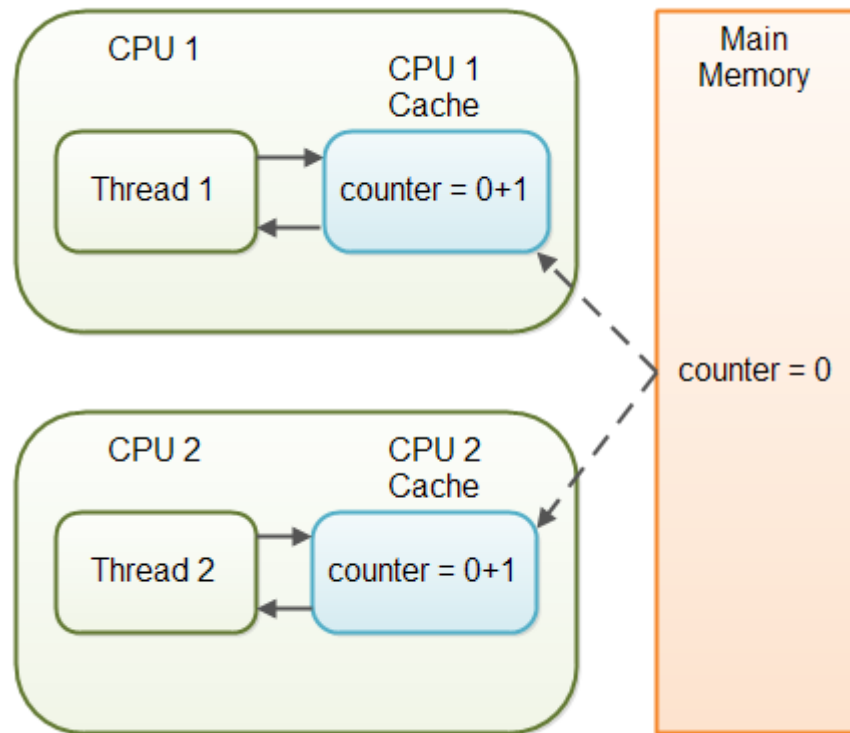
Volatile nem sempre é suficiente

- Mesmo que o **volatile** garanta que todas as operações de leituras e escrita sejam feitas diretamente na memória principal, ainda existem situações que isso não é o bastante.
- No primeiro exemplo, a variável **counter** é modificada pela Thread 1. Declarar essa variável como volatile é suficiente para que a Thread 2 acesse seu valor atualizado.
- Várias threads podem escrever na variável **counter** ao mesmo tempo. No entanto, o valor de counter depende de uma leitura prévia.

Volatile nem sempre é suficiente

- Entre o tempo de leitura e escrita existe um pequeno “gap” entre o tempo que uma thread lê counter e depois escreve seu novo valor na memória principal.
- Esse “gap” cria uma condição de corrida, onde várias threads (em um caso mais raro), podem ler a variável da memória principal ao mesmo tempo, atualizar seu valor, e escrevê-lo na memória principal, sobrescrevendo os valores calculados por outras threads.

Volatile nem sempre é suficiente



Quando então é suficiente????

- Se duas threads estão lendo e escrevendo em uma mesma variável compartilhada, o uso apenas de volatile não garante um bom funcionamento.
- A operação deve ser **atômica**, e isso é conseguido através do **synchronized**. O volatile não bloqueia o acesso de threads.
- Se apenas uma única thread lê e escreve na variável compartilhada, enquanto as outras threads apenas lêem, daí sim, a threads leitoras estão garantidas de sempre verem o valor mais atualizado. Caso a variável não fosse volatile, isso não seria garantido.

Referência

- <http://tutorials.jenkov.com/java-concurrency/volatile.html>