Programação Concorrente em Java

Aula 07

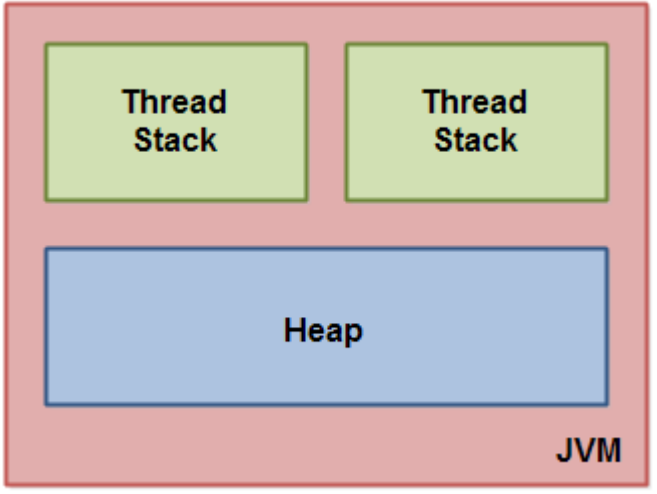
Modelo de Memória do Java

Introdução

- O modelo de memória do especifica como a JVM conversa com a memória RAM do computador, via sistema operacional.
- A JVM (Java Virtual Machine) é um modelo inteiro de um computador, logo, inclui também uma memória interna (Java MemoryModel).
- É importante entender o modelo de memória do Java para poder construir programas concorrentes.

Java Memory Model

- A JVM divide a memória entre thread stacks e a heap.



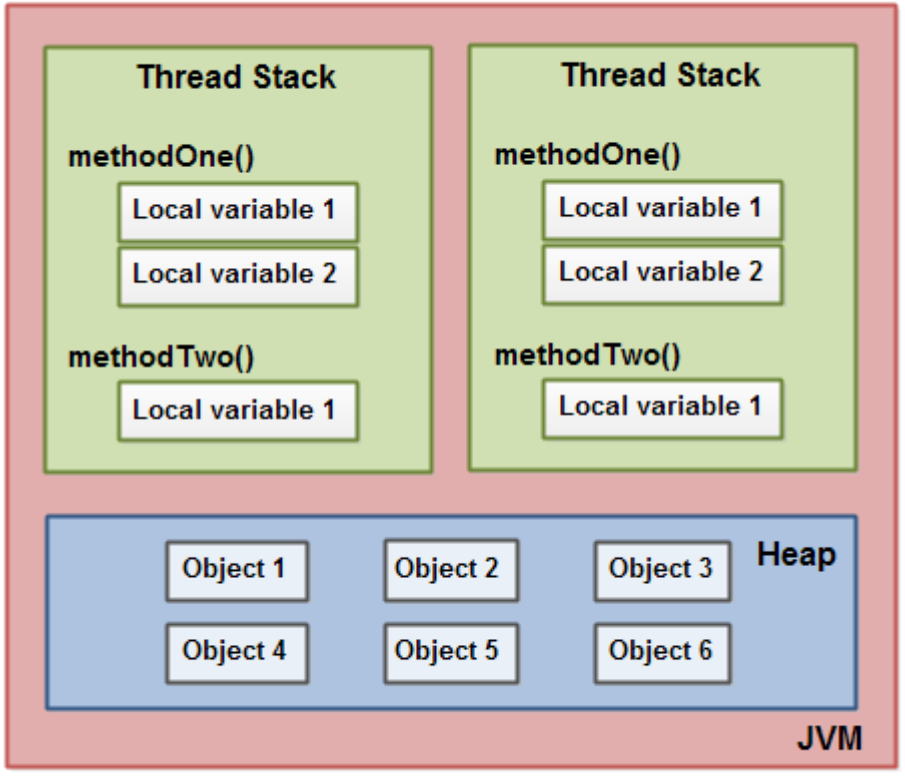
Java Memory Model

- Cada thread em execução tem sua própria stack
  - A stack contem informações sobre os métodos que a thread chamou (call stack).
  - A medida que a thread vai executando, a call stack muda.
  - A call stack também inclui as variáveis locais para cada método em execução.
  - Variáveis primitivas são também armazenadas na call stack e cada thread tem a sua pória cópia.

Java Memory Model

- A heap (monte) contem todos os objetos criados pela aplicação.
- Não importa se um objeto foi criado e atribuído a uma variável local ou criado como uma variável membro de outro objeto.

Java Memory Model



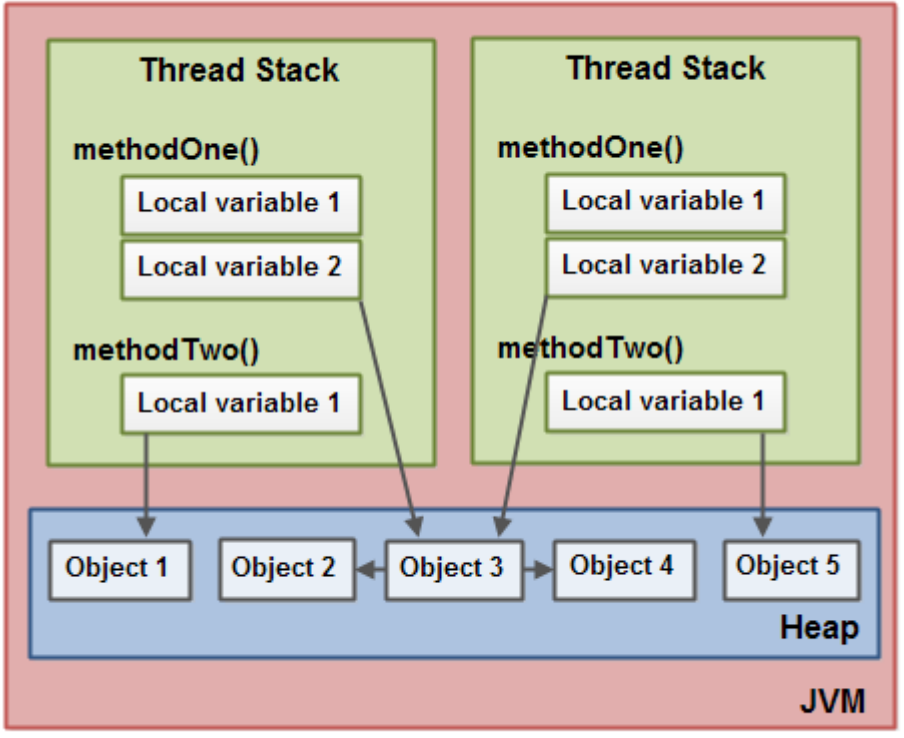
Java Memory Model

- Uma variável local pode ser um tipo primitivo.
- Uma variável local pode também ser uma referência para um objeto (a referência fica na call stack mas o objeto fica no heap).
- Um objeto pode conter métodos e esses métodos, variáveis locais. Essas variáveis locais são armazenadas na thread stack, mesmo o objeto estando na heap.
- Membros de objeto são também armazenados na heap, junto com os objetos os quais pertencem. Isso é verdade para tipos primitivos e referências para outros objetos.

Java Memory Model

- Variáveis de classe (estáticas) também são armazenadas na heap.
- Objetos na heap podem ser acessados por qualquer thread que tenha uma referência ao objeto.
- Quando uma thread tem acesso a um obejto, via referência, a thread também pode ter acesso aos membros do objeto (via método, por exemplo...).

Java Memory Model

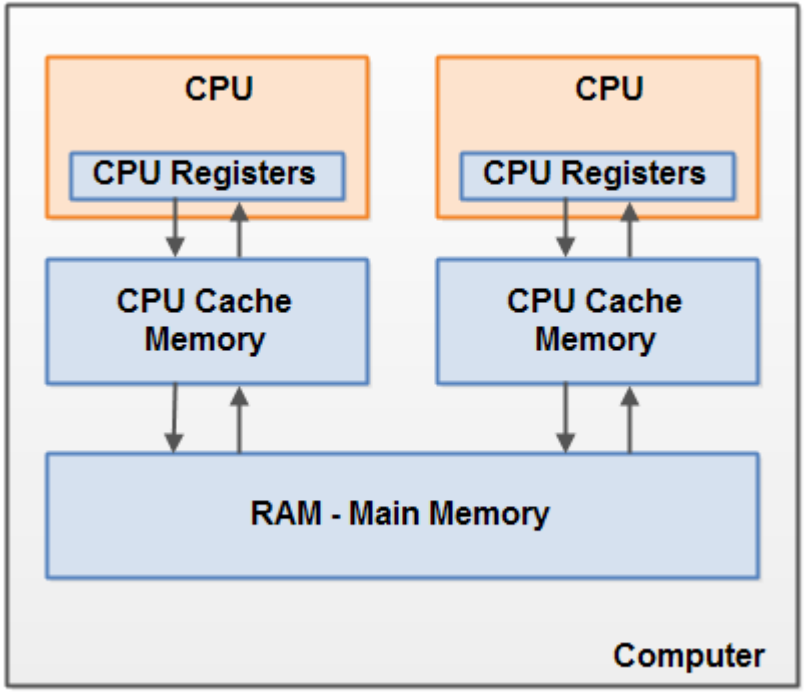


Java Memory Model

```
public class MyRunnable implements Runnable() {
    public void run() {
        methodOne();
    }
    public void methodOne() {
        int localVariable1 = 45;
        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;
        //... do more with local variables.
        methodTwo();
    }
    public void methodTwo() {
        Integer localVariable1 = new Integer(99);
        //... do more with local variable.
    }
}
```

```
public class MySharedObject {
    //static variable pointing to instance of MySharedObject
    public static final MySharedObject sharedInstance =
        new MySharedObject();
    //member variables pointing to two objects on the heap
    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);
    public long member1 = 12345;
    public long member1 = 67890;
}
```

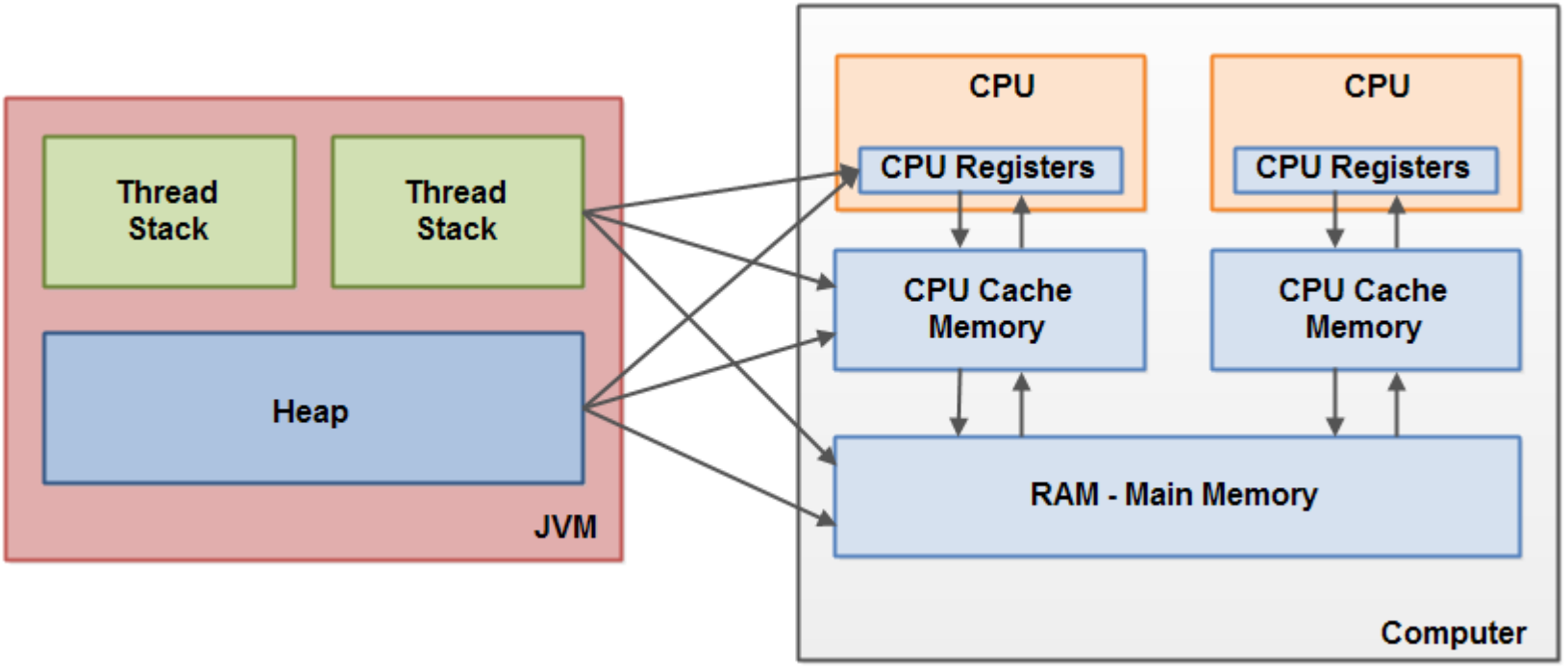
Arquitetura de Hardware da Memória



Arquitetura de Hardware da Memória

- Um computador moderno tem geralmente duas ou mais CPUs. Em alguns cados, CPUs podem ter vários cores.
- Sendo assim, computadores modernos podem ter mais de uma thread realmente executando ao mesmo tempo.
- Cada CPU tem um conjunto de registradores, que funcionam como uma memória interna da CPU, agilizando os cálculos.
- Além disso, CPUs tem memória cache, de diversos tamanhos e níveis (L1, L2, L3). A mémoria cache é bem mais rápida que a memória RAM mas não tanto quanto os registradores.
- “Cache lines”

Java VS Hardware



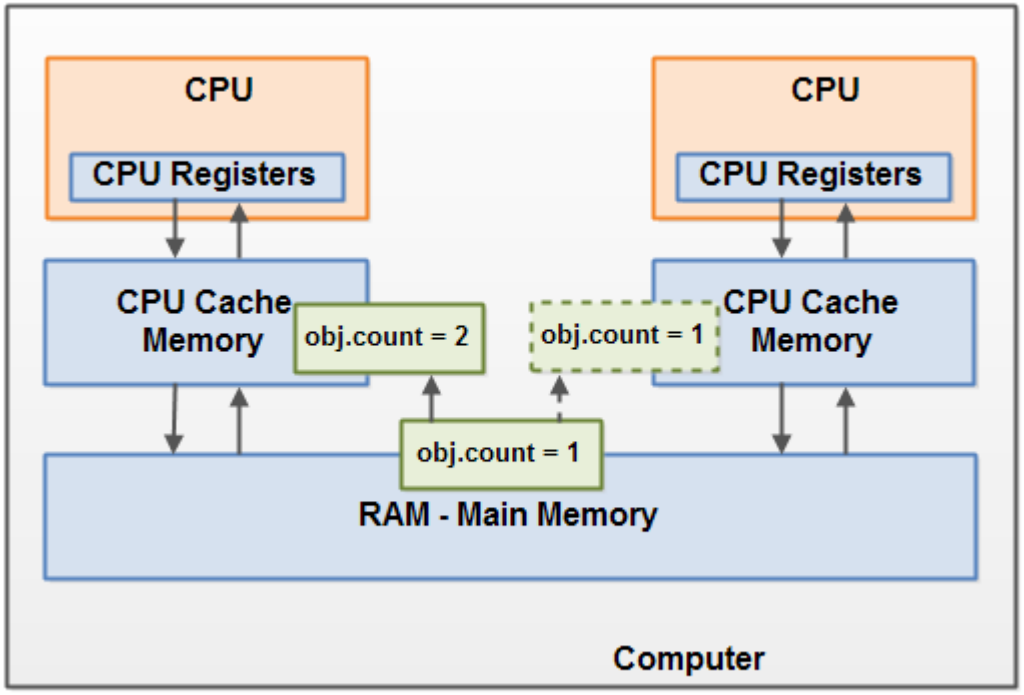
Java VS Hardware

- Quando objetos podem ser armazenados em diferentes áreas da memória, certos problemas podem acontecer:
  - Visibilidade de atualizações de threads em variáveis compartilhadas.
  - Condições de corrida

Visibilidade de objetos compartilhados

- Imagine que um objeto compartilhado é armazenado na memória RAM.
- Uma thread executando na CPU 1 lê o objeto e armazena em sua cache (CPU 1). A thread faz uma modificação no objeto mas a operação flush (atualizar a memória principal) ainda não foi disparada. O objeto está modificao apenas na cache da CPU 1.
- Desta forma, cada thread pode acabar com a sua própria cópia do objeto nas memórias caches de outras CPUs.
- Para resolver esse problema deve-se usar a palavra reservada do Java **volatile.**
- Variáveis voláteis só pode ser lidas diretamente da memória e sempre são escritas diretamente na memória, quando **atualizadas**.

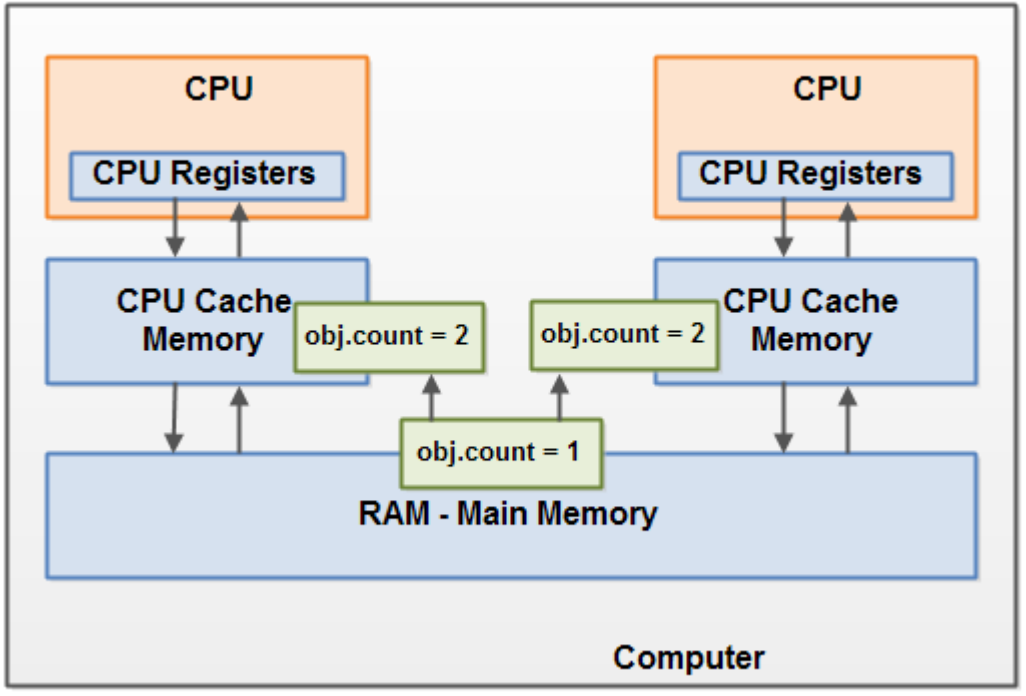
Visibilidade de objetos compartilhados



Condições de Corrida

- Imagine uma thread A lê a variável count de um objeto compartilhado no cache de sua CPU. Thread B faz a mesma coisa (em uma CPU diferente). Agora, thread A adiciona 1 ao count e thread B faz a mesma coisa.
- Qual valor será escrito na memória, no final da aplicação?
- Para resolver esse problema, deve-se fazer uso da palavra reservada **synchronized, dentro de um bloco,** do Java.

Condições de Corrida



Referências

- http://tutorials.jenkov.com/java-concurrency/java-memory-model.html