

hulu

Thread-safe programming on Java Memory Model

Let's begin with a question...

无奖问答:如果有两个线程同时执行如下read函数和write函数, read函数的执行结果 (r1, r2) 可能是?

```
Class Demo {  
    int x = 0, y = 0;  
  
    public void write() {  
        x = 1;          // L1  
        y = 2;          // L2  
    }  
  
    public void read() {  
        int r1 = y;    // L3  
        int r2 = x;    // L4  
    }  
}
```

hulu

✓ A. (0, 0)

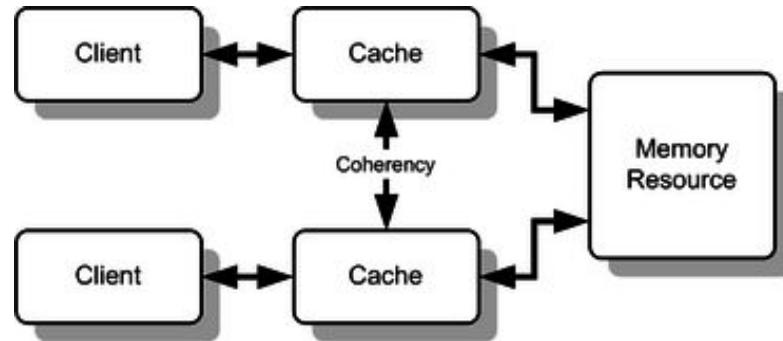
✓ B. (0, 1)

✓ C. (2, 0) ➡ where reordering(a.k.a visibility,
memory effect) happens!

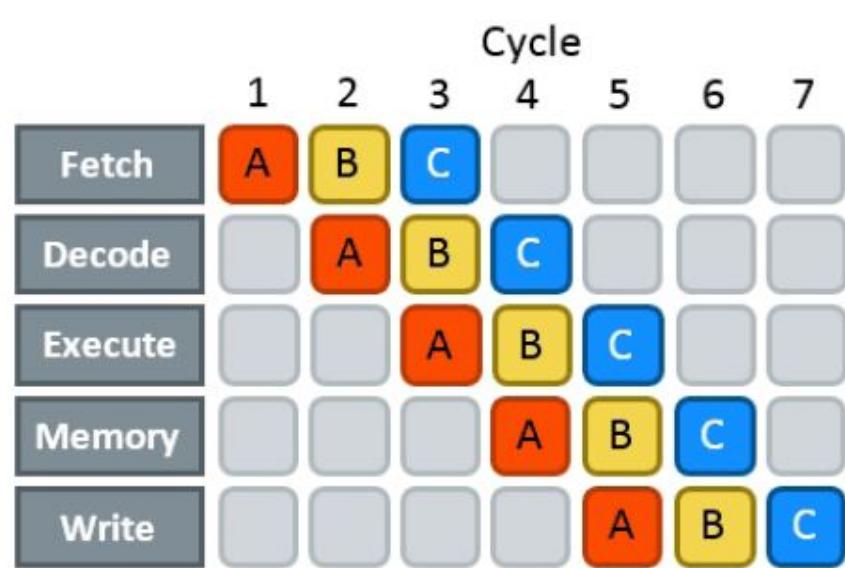
✓ D. (2, 1)

Why can reordering happen?

1. 缓存系统导致的reordering



2. CPU为了流水线效率产生的reordering



3. 编译器优化导致的reordering(如重排序、使用寄存器等)

Big Problems

1. 禁止reordering会导致极大的性能损失, 以至于是不现实的
2. Reordering随体系结构、编译器(JVM)的实现不同而不尽相同

Now, what's the Java Memory Model?

Java Memory Model(JMM)提供了一套统一的规范，用于约束Java程序的reordering行为。

The Java programming language memory model works by examining each read in an execution trace and checking that the write observed by that read is valid according to certain rules*.

The Goal of Java Memory Model

- JMM需要允许一定限度的reordering，以确保一些普遍的体系结构和编译器优化技术得以运用
- JMM需要定义必须的同步操作，“正确同步”的程序不会由于reordering导致错误
- 对于未“正确同步”的程序，JMM需要列明其可能产生的reordering错误，以评估可能的程序危害

The Main Components of Java Memory Model

- *Happens-before* relationship
- Read Rule
- *Final* Semantics



Some notes before diving into...

- As-if-serial - 单线程程序不会由于reordering导致错误。也可以说, reordering指一个线程的执行顺序在另一个线程眼中是乱序的。
- Reordering只发生于对线程间共享变量的操作(包括object field, static field, array element), 对于局部变量不存在reordering问题。

```
Class Demo2 {  
    int x = 0, y = 0;  
  
    public void write() {  
        x = 2;  
        y = x + 1;  
        int r1 = x;  
        int r2 = y;  
    }  
}
```

hulu

As-if-serial Example: (r1, r2) is always (2, 3)

Happens-before Relationship

程序操作(语句)根据代码顺序和同步操作顺序形成的一种偏序关系
，主要包括：

- 如果操作A的代码顺序先于操作B，则操作A **happens-before** 操作B
- 对一个object的monitor unlock操作(sychronized block末尾处) **happens-before** 对该object后续的monitor lock操作(synchronized block起始处)
- 对一个volatile variable的写操作 **happens-before** 对该object后续的读操作
- 对一个线程的Thread.start()操作 **happens-before** 该线程内的所有操作
- 一个线程内的所有操作 **happens-before** 对该线程的Thread.join()操作
- 如果存在操作A **happens-before** 操作B，且操作B **happens-before** 操作C，则操作A **happens-before** 操作C

★ 如果操作A **happens-before** 操作B，则可认为操作A先于操作B执行

Thread 1	Thread 2
<pre>synchronized { x = 1; // L1 y = 2; // L2 }</pre>	<pre>synchronized { int r1 = y; // L3 int r2 = x; // L4 }</pre>

此时，因为存在L1 *hp* L2, L3 *hp* L4, L2 *hp* L3
所以 L1 *hp* L2 *hp* L3 *hp* L4
r1 = 2, r2 = 1

Read Rule

对于一个共享变量的读操作R和写操作W, R读到W的值的需要满足以下条件之一

1. 如果存在 $W \text{ hp } R$, 则不存在其他的 W' , 满足 $W \text{ hp } W'$ 且 $W' \text{ hp } R$
2. W和R不存在hp关系, 此时R允许读到W的值

Thread 1	Thread 2
<pre>x = 1; // L1 y = 2; // L2</pre>	<pre>int r1 = y; // L3 int r2 = x; // L4</pre>

此时L3和L2没有hp关系, 则L3可以读到L2的值(2)或是初始值(0)
L1和L4同理

Final Semantics

```
public class Demo2 {  
    static Demo2 value;  
    private int x;  
    private int y;  
  
    public Demo2 {  
        x = 4;  
        y = 5;  
    }  
  
    public static void actor1() {  
        value = new Demo2();  
    }  
  
    public static void actor2() {  
        Demo2 current = value;  
        int r;  
  
        if (current != null) {  
            r = current.x + current.y;  
        } else {  
            r = -1;  
        }  
    }  
}
```

{使用final可能遇到哪些问题？如何避免？
如何避免final int x...}

对于Final字段及其所引用的对象(object或array)的访问确保能得到最新的值。

！例外：如果该变量在初始化的过程中被泄漏(直接或间接)，则无法保证访问到最新值。

```
public Broken {  
    public static Broken value;  
  
    public final int f;  
  
    public Broken() {  
        f = 100;  
        value = this;  
    }  
  
    public static read() {  
        if (value != null) {  
            int r = value.f;          读取r可能得到0！  
        }  
    }  
}
```

The Double-checked Locking Pattern

该程序会不会产生并发问题？如果有的话怎么修改？

```
private static Something instance = null;

public Something getInstance() {
    if (instance == null) {
        synchronized (this) {
            if (instance == null)
                instance = new Something();
        }
    }
    return instance;
}
```

Option 1 - 使用volatile关键字

```
private static volatile Something instance = null;
```

Option 2 - 使用on-demand holder pattern

```
private static class LazySomethingHolder {
    public static Something something = new Something();
}

public static Something getInstance() {
    return LazySomethingHolder.something;
}
```

Correctly Synchronized Program

- Data Race - 对于一个共享变量的两个操作A, B, 如果满足
 - 至少有一个是写操作
 - A和B不存在 hp 关系
- 如果在程序的所有执行中都不存在data race, 则该程序是correctly synchronized。
- Correctly synchronized程序不会由于reordering产生错误。(但依然要考虑其他并发问题, 如原子性)
- Java Concurrency Stress Test(jcstress) - 用于测试Java并发程序正确性的框架
 - [例子](#)

Summary: Some Best Practices of Concurrent Programming

- 避免数据竞争。
- 使用synchronized/volatile确保程序正确的memory effect语义。
- 使用final关键字定义共享字段。
- 在线程开始前初始化共享变量。
- 避免object初始化时引用泄漏
 - 被赋值到外部field
 - 被注册为listener/hook
 - 通过inner class被隐式地泄漏
- 正确实现double checked locking。
- Volatile可用作一种更轻量的同步机制，它具备和synchronized相同的memory effect语义，但避免了锁竞争。

hulu

THANK YOU