JMM
Java Memory Model

黄智辉

Agenda

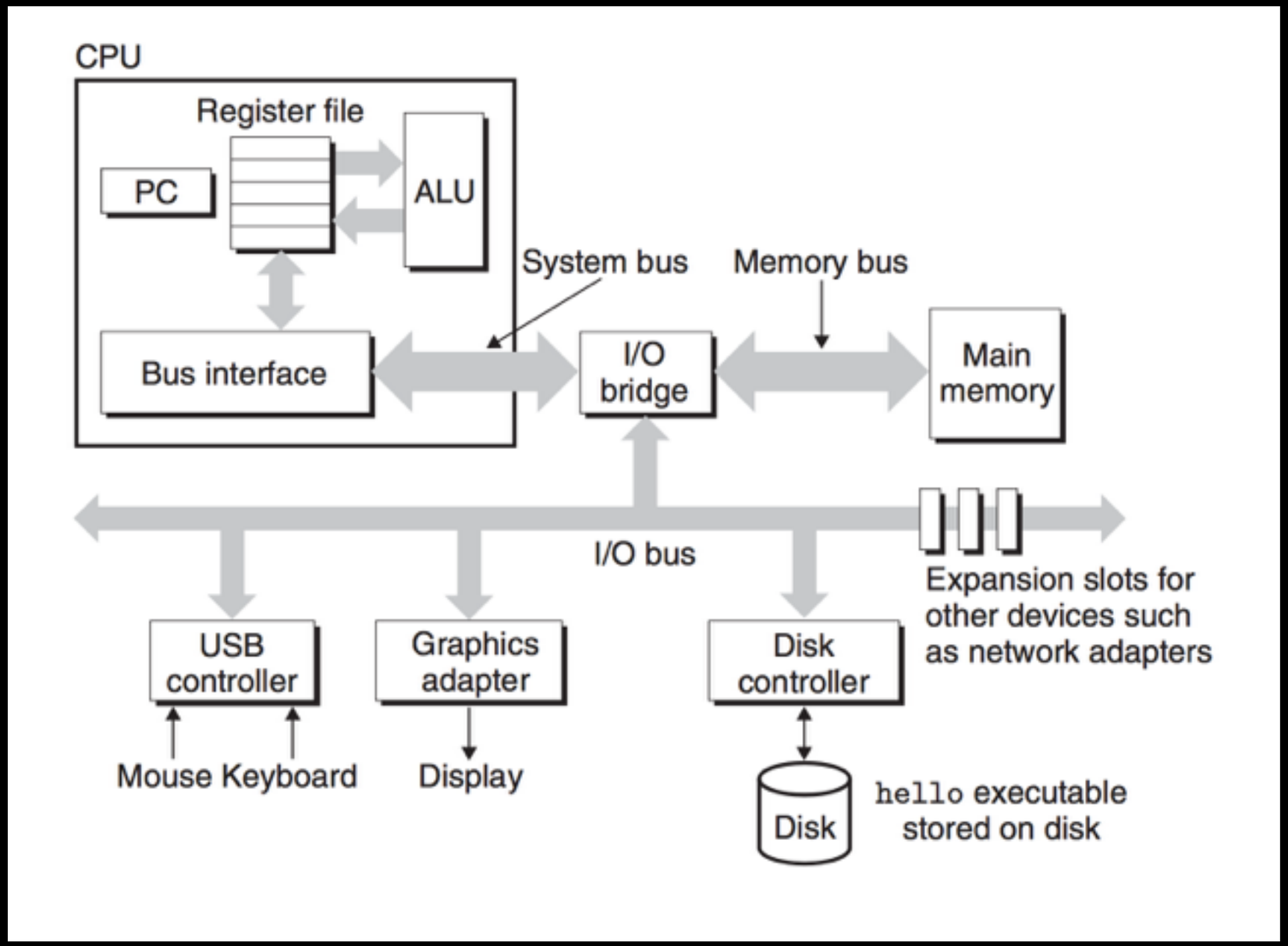
- 为什么需要JMM
- JMM是什么?
- JMM举例
- 并发编程中的经典问题

为什么需要JMM?

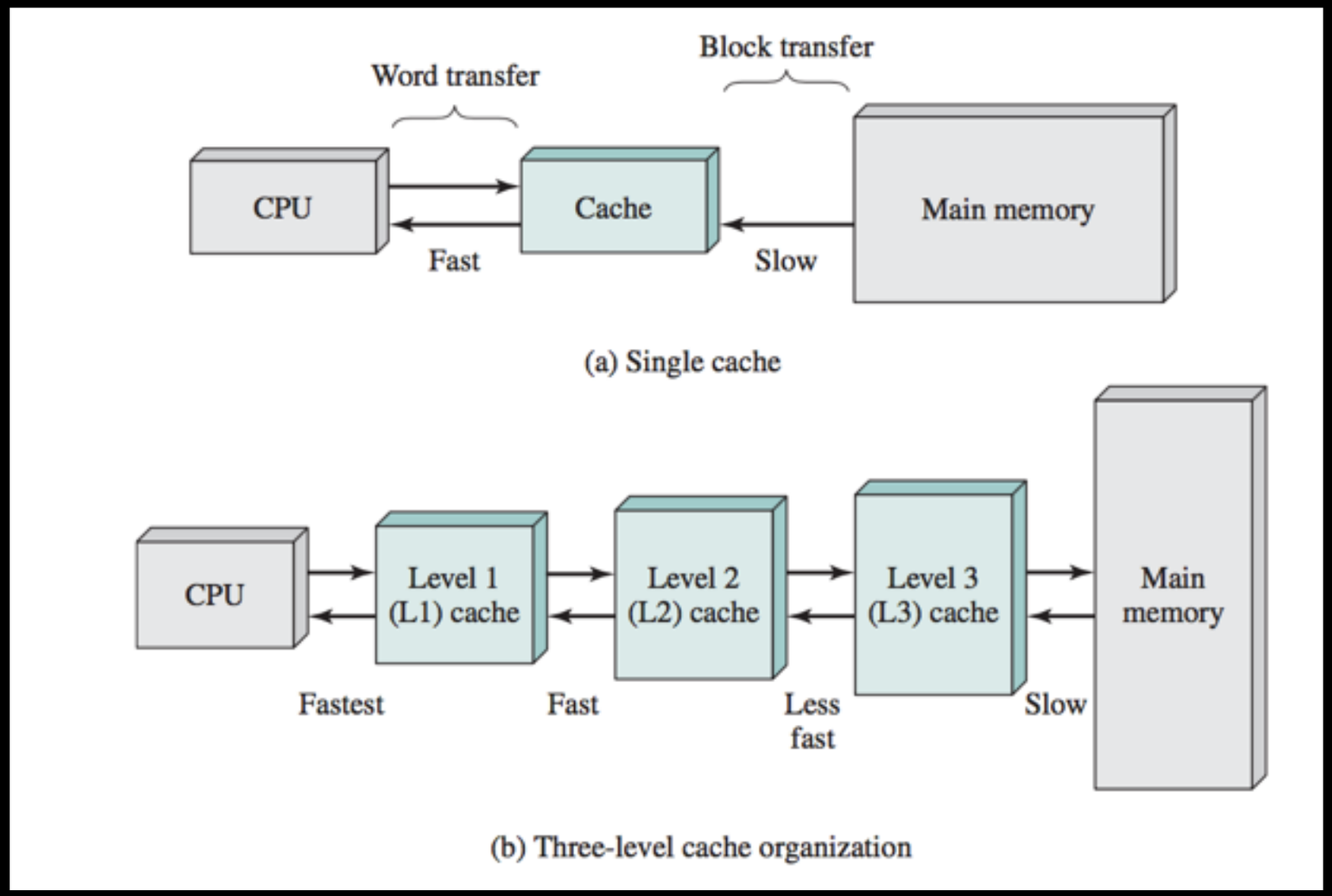
该程序执行的最终结果可能是?

```
public class PossibleRecording {

    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException{

        Thread one = new Thread(new Runnable() {
            @Override
            public void run() {
                a = 1;  // A1
                x = b;  // A2
            }
        });
        Thread two = new Thread(new Runnable() {
            @Override
            public void run() {
                b = 2;  //B1
                y = a;  //B2
            }
        });
        one.start();
        two.start();
        one.join();
        two.join();
        System.out.println("x =" + x + ";" + "y =" + y);
    }
}
```

计算机是怎样执行指令的?

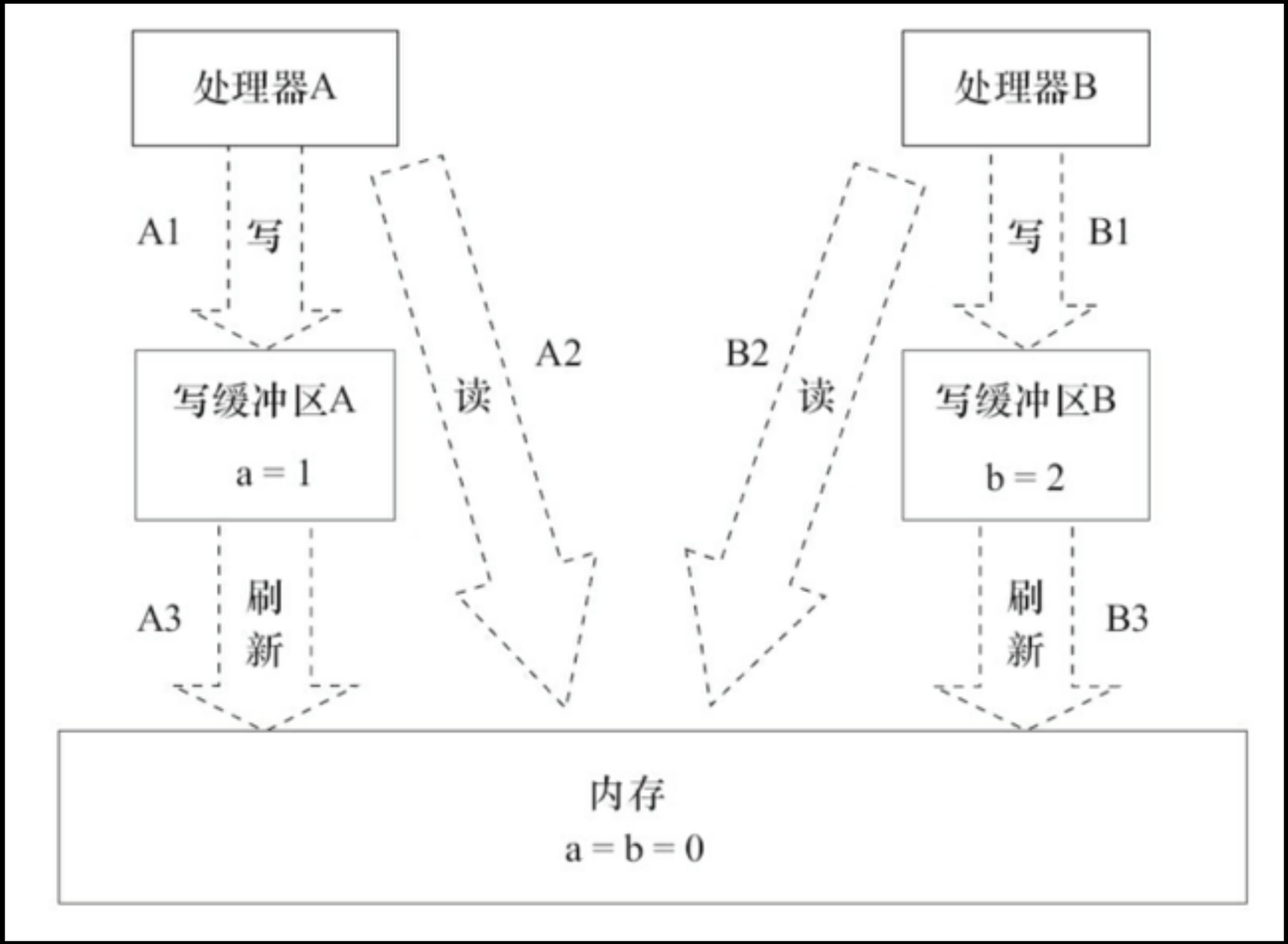


CPU的三级缓存



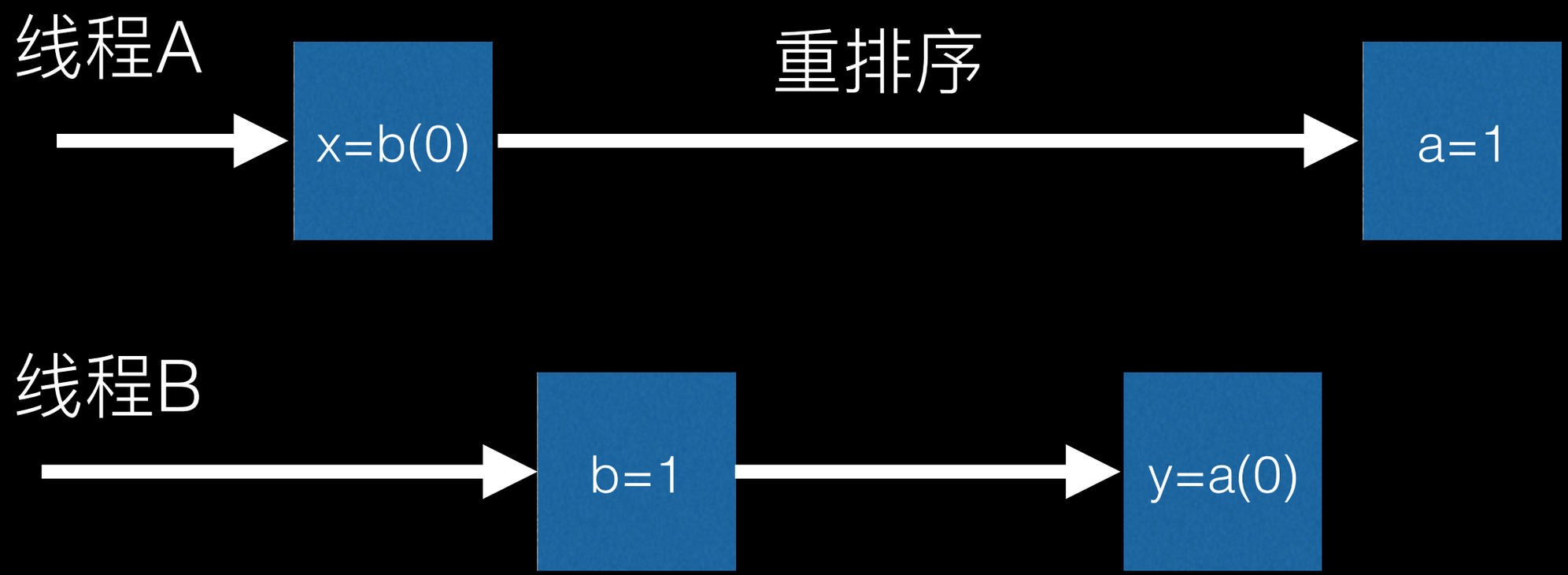
为什么需要三级缓存?

出现x= 0; y= 0的原因



实际的效果就是A1和A2，B1和B2重排序了

出现x= 0; y= 0的原因



为什么需要JMM?

指令重排序所带来的问题

各种处理器的内存重排序规则

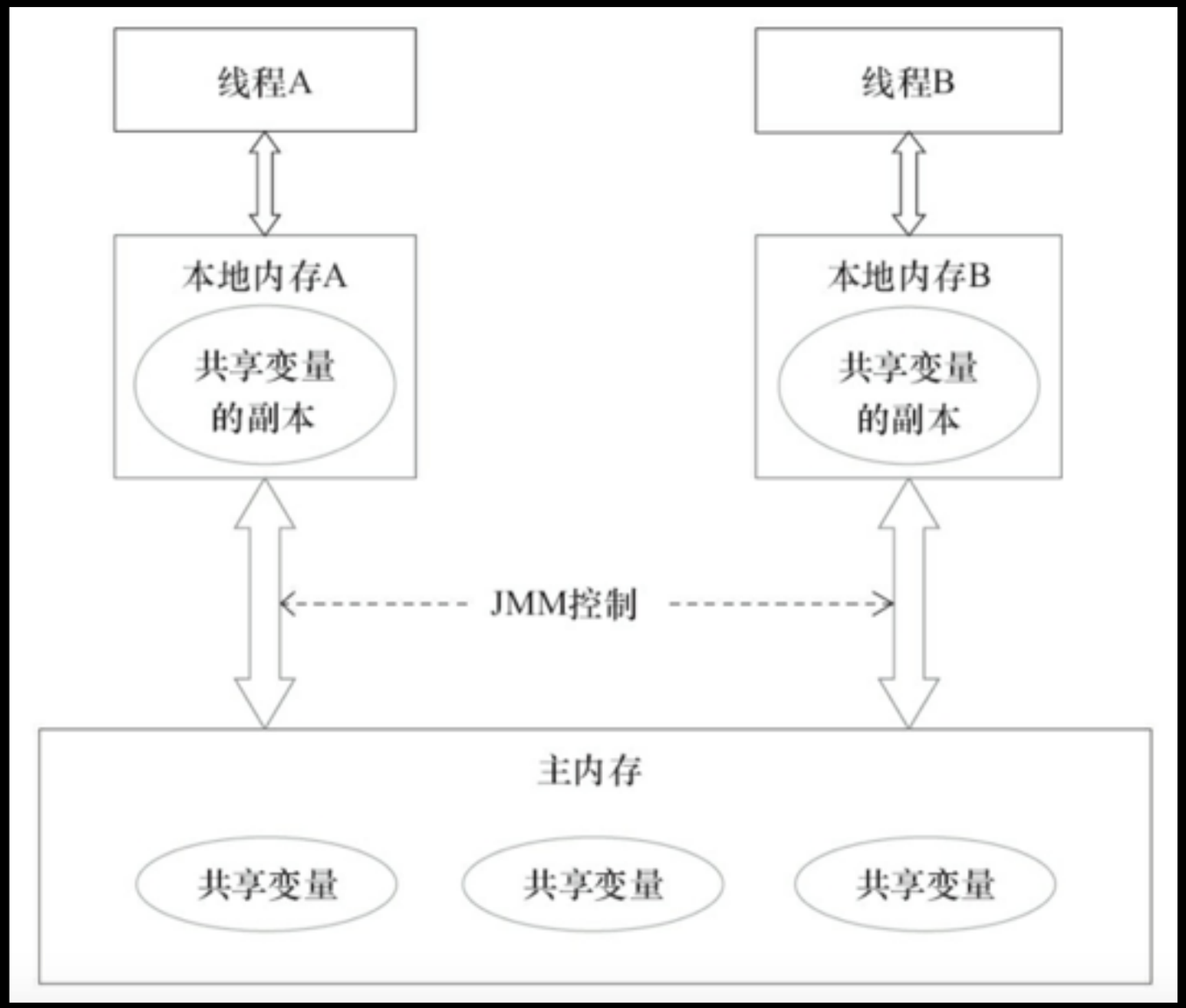
规则 / 处理器	Load-Load	Load-Store	Store-Store	Store-Load	数据依赖
SPARC-TSO	N	N	N	Y	N
x86	N	N	N	Y	N
IA64	Y	Y	Y	Y	N
PowerPC	Y	Y	Y	Y	N

两个线程同时操作时i最后得多少?

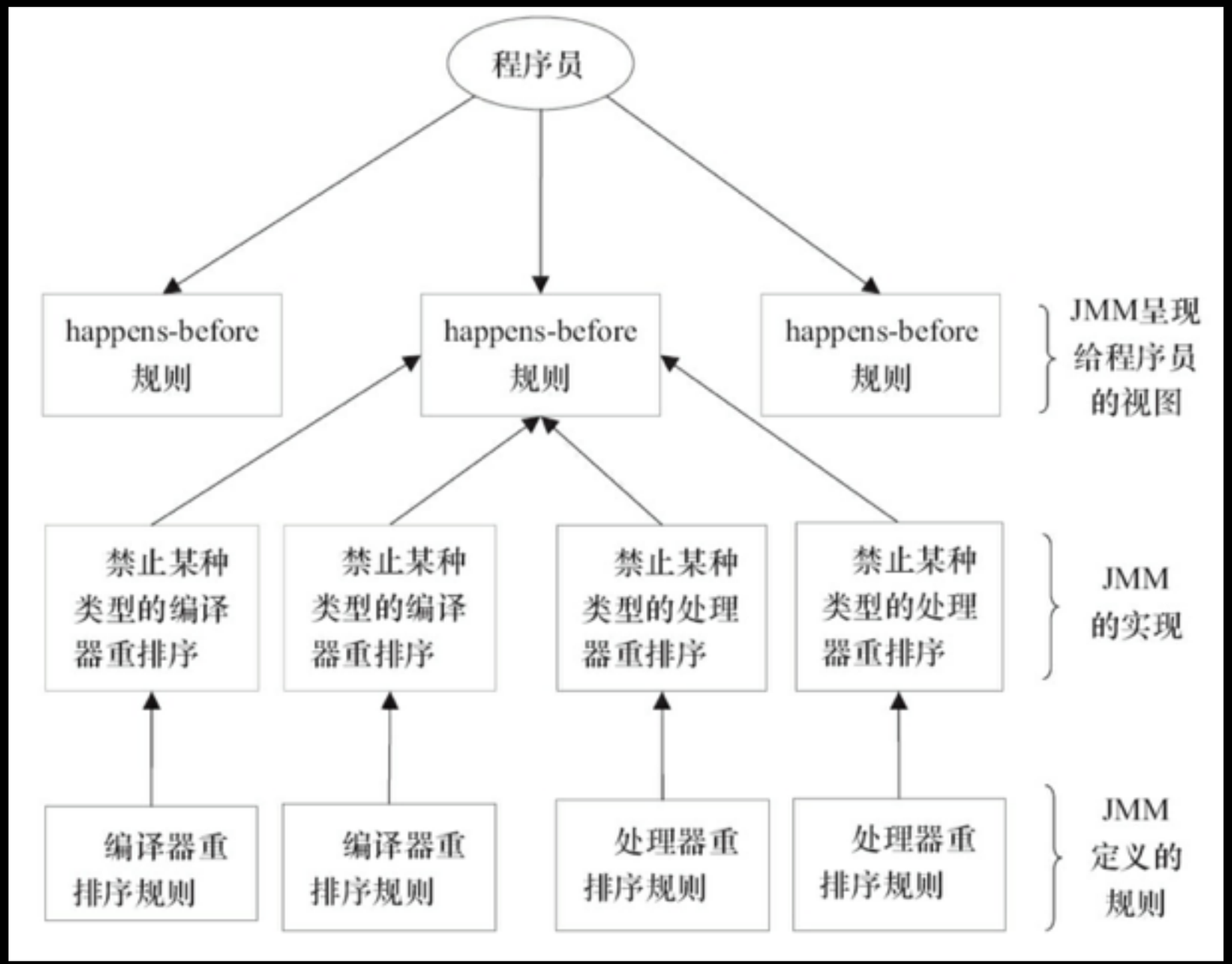
```
public class RecorderExample {

    int a = 0;
    boolean flag = false;
    public void write() {
        a = 1;              // 1
        flag = true;    // 2
    }
    public void reader() {
        if (flag) {       // 3
            int i = a * a; // 4
            System.out.println("i ==" + i);
        }
    }
}
```

JMM是什么?

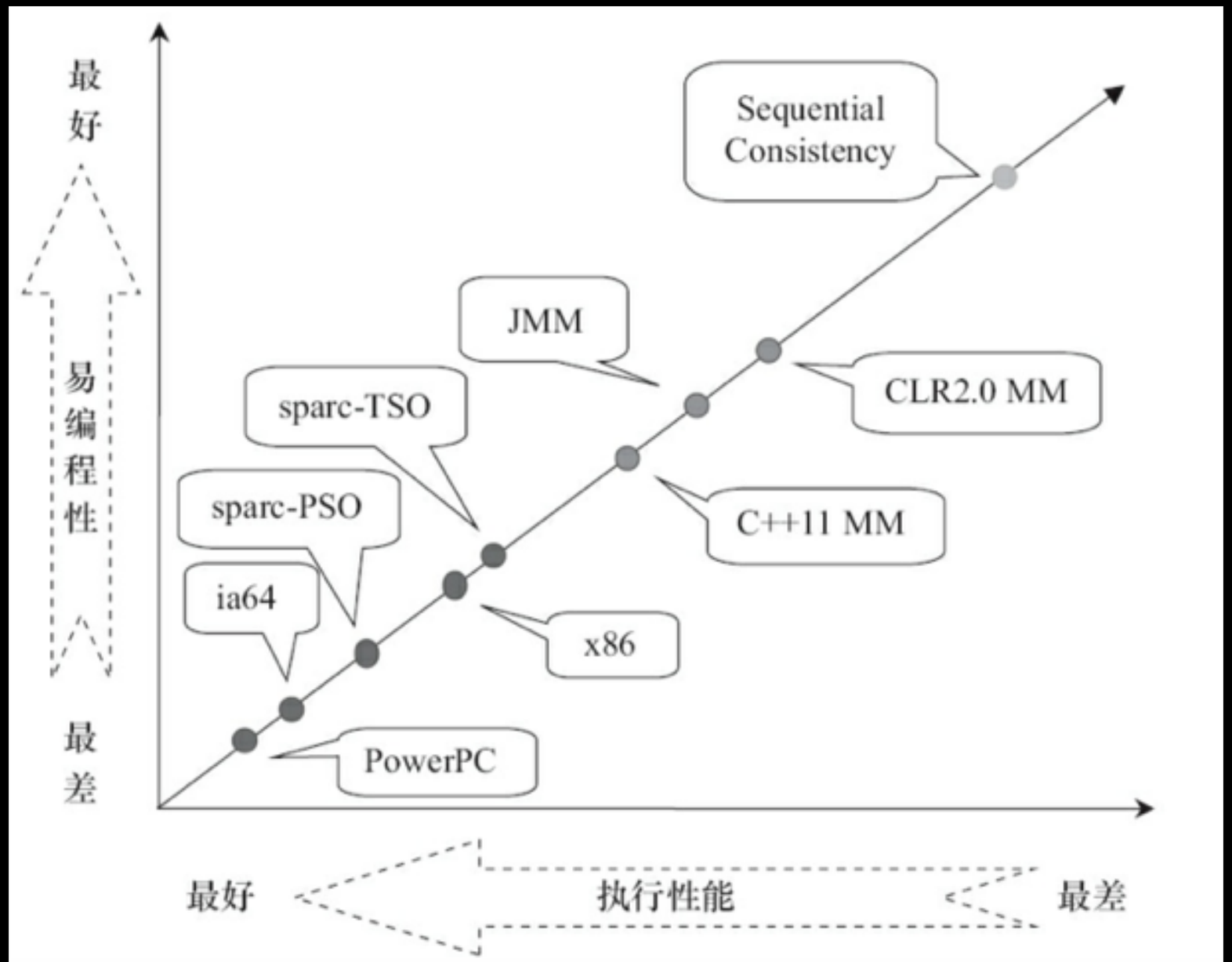
JMM是控制一个线程对共享变量的写入何时对另外一个线程可见的一种机制。



JMM和Happens-Before规则



各种内存模型之间的对比

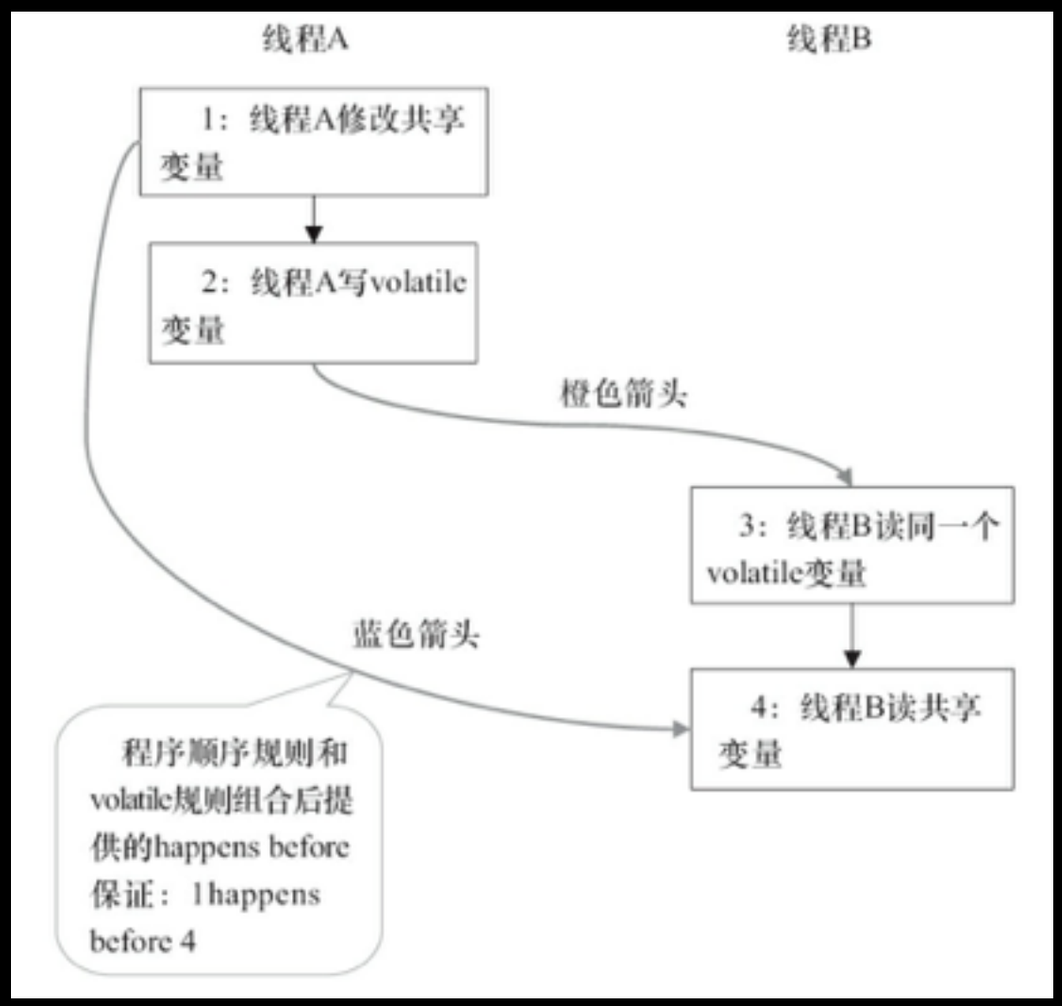


JMM是如何实现避免指令重排序的?

屏障类型	指令示例
LoadLoad Barriers	Load1; LoadLoad; Load2
StoreStore Barriers	Store1; StoreStore; Store2
LoadStore Barriers	Load1; LoadStore; Store2
StoreLoad Barriers	Store1; StoreLoad; Load2

Volatile的内存语义

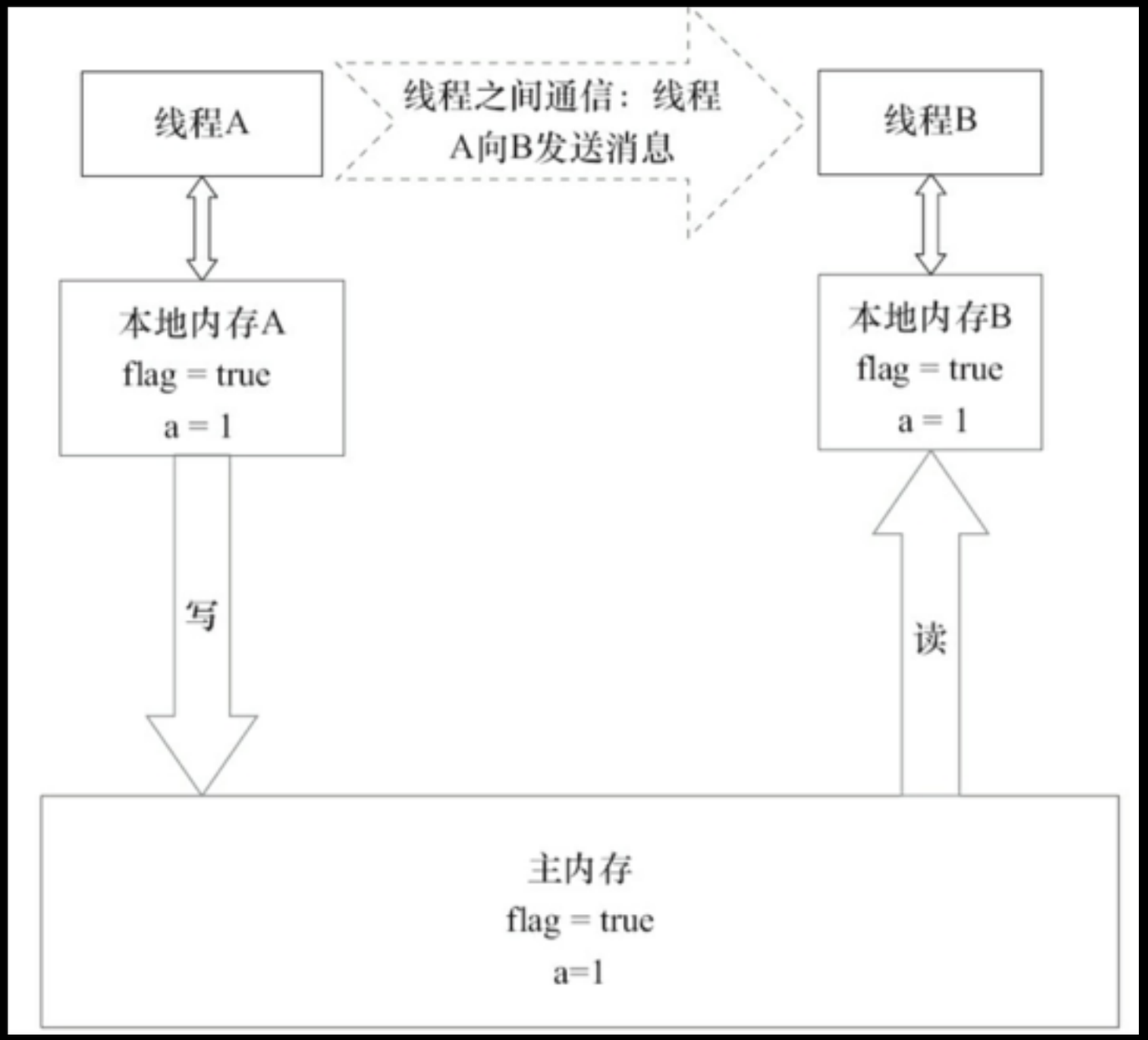
```java
public class VolatileExample {

    int a = 0;
    volatile boolean flag = false;

    public void writer() {
        a = 1;    // 1
        flag = true; // 2
    }

    public void reader() {

        if (flag) { // 3

            int i = a; // 4
            //.....
        }
    }
}
```




```java
AtomicInteger atomicInteger = new AtomicInteger( initialValue: 3);
if (atomicInteger.get() == 3) {
    atomicInteger.set(4);
}
```

volatile++这种操作是线程安全的吗?

Volatile内存语义的实现



指令执行顺序

普通读

普通写

StoreStrore屏障

volatile写

StroreLoad屏障

Final的内存语义

```
public class FinalExample {

    int i; // 普通变量
    final int j; // final变量
    static FinalExample obj;

    public FinalExample() { //构造函数
        i = 1;                  //写普通域
        j = 2;                  //写final域
    }
    public static void writer() { // 写线程执行

        obj = new FinalExample();
    }
    public static void reader() { //读线程B执行

        FinalExample object = obj; //读线程引用
        int a = object.i;          //读普通域
        int b = object.j;          //读final域
    }
}
```

线程A

线程B

构造函数开始

写final域: j = 2

StroreStore屏障

构造函数结束

构造对象赋值给obj

写普通域:
i = 1

读对象应用obj

读对象普通域i

读对象的final域j

时间

CAS例子

- 代码实例

CAS的三大问题

- ABA问题
- 循环时间长开销大
- 只能保证一个共享变量的原子操作

ABA问题

- ABA问题的由来
- 怎样解决ABA问题

Amdahl定律

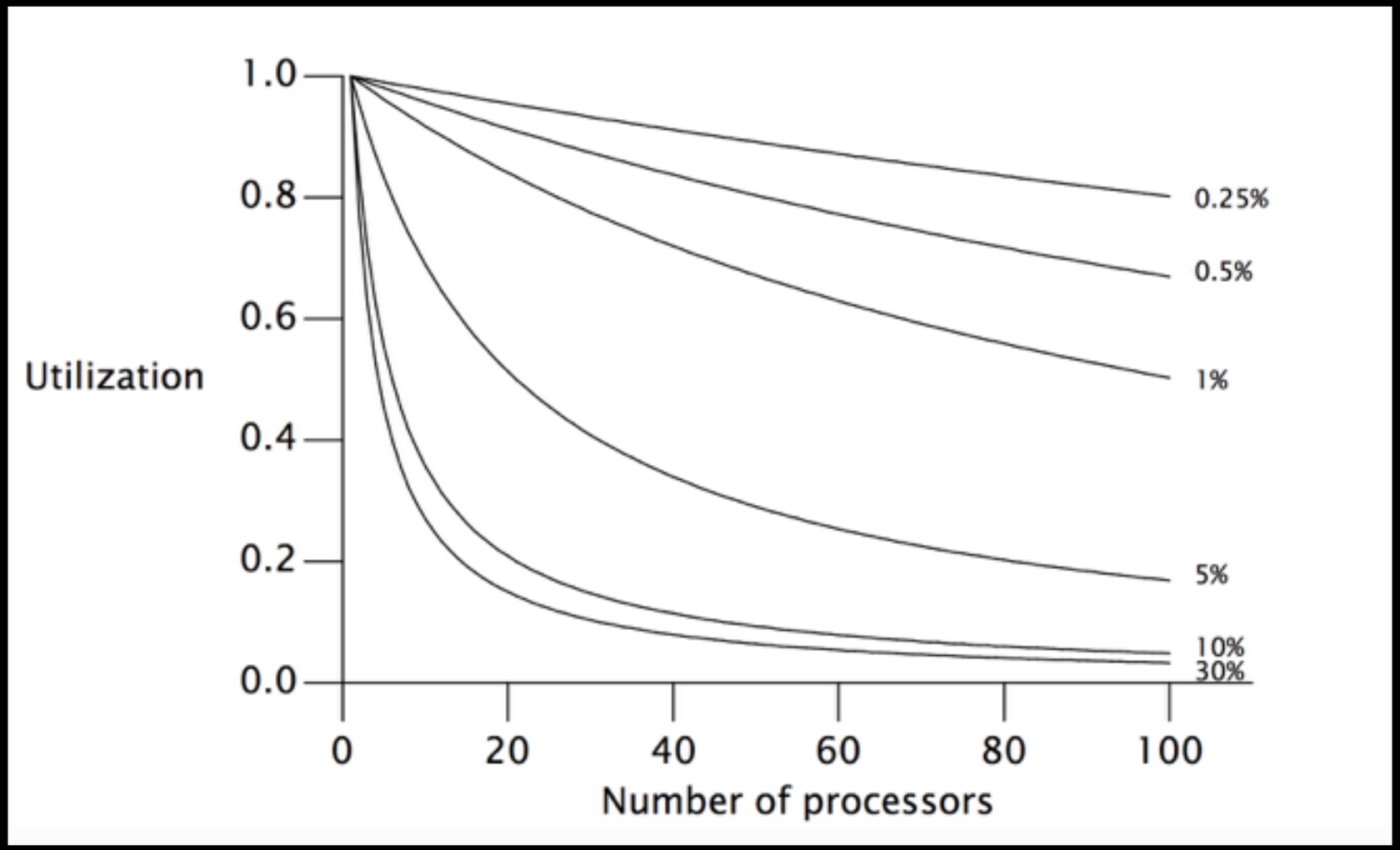
$$Speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

F: 串行部分所占的百分比

N: 处理器的个数

增加处理器的个数一定能提升程序的执行速度吗?

在一定的串行比率情况下：
处理器的个数和资源利用率之间关系



双重锁定问题

- 代码实例

volatile解决方案

- 代码实例

初始化安全问题

- 代码实例

延迟初始化方案小结

如果需要对实例字段使用线程安全的延迟初始化，利用volatile。
如果需要对静态字段使用线程安全的延迟初始化，适应基于类的初始化方案

Happens-before规则

- 程序顺序规则
- 监视器锁规则
- volatile变量规则
- 传递性
- start()规则
- join()规则

参考资料

- 深入理解计算机操作系统
- Java并发编程实战
- Java并发编程的艺术
- 深入浅出JCU
- 操作系统—精髓与设计原理
- 编码—隐藏在计算机软硬件背后的语言