曹大谈内存重排

目录

写这篇文章的原因很简单，公司内部的 Golang 社区组织了第一期分享，主讲嘉宾就是我们敬爱的曹大。这个必定是要去听的，只是曹大的讲题非常硬核，所以提前找他要了参考资料，花了 1 个小时提前预习，才不至于在正式分享的时候什么也不懂。当然了，这也是对自己和主讲者的尊重。所有的参考资料都在文章最后一部分，欢迎自行探索。

在我读曹大给我的中英文参考资料时，我发现英文的我能读懂，读中文却很费劲。经过对比，我发现，英文文章是由一个例子引入，循序渐进，逐步深入。跟着作者的脚步探索，非常有意思。而中文的博客上来就直奔主题，对于第一次接触的人非常不友好。

两者就像演绎法和归纳法区别。国内的教材通常是演绎法，也就是上来先讲各种概念、原理，再推出另一些定理，比较枯燥；国外的教材更喜欢由例子引入，步步深入，引人入胜。这里，不去评判孰孰劣。多看看一些英文

原版材料，总是有益的。据我所知，曹大经常从亚马逊上购买英文书籍，这个侧面也可以反映曹大的水平高啊。据说英文书一般都很贵，可见曹大也是很有钱的。

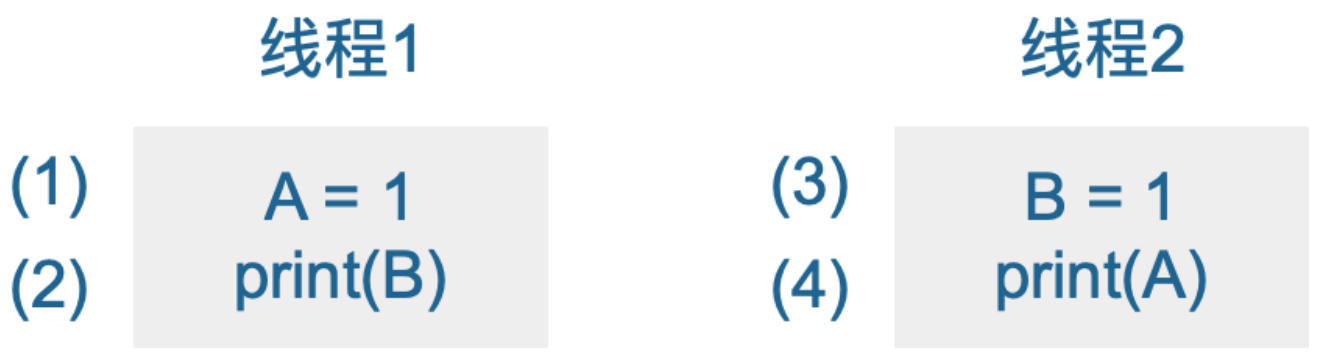
所以啊，技术文章写好不容易，我也自省一下。

什么是内存重排

分两种，硬件和软件层面的，包括 CPU 重排、编译器重排。

CPU 重排

引用参考资料 【内存一致模型】 里的例子：



在两个线程里同时执行上面的代码，A 和 B 初始化值都是 0，那最终的输出是什么？

先说几种显而易见的结果：

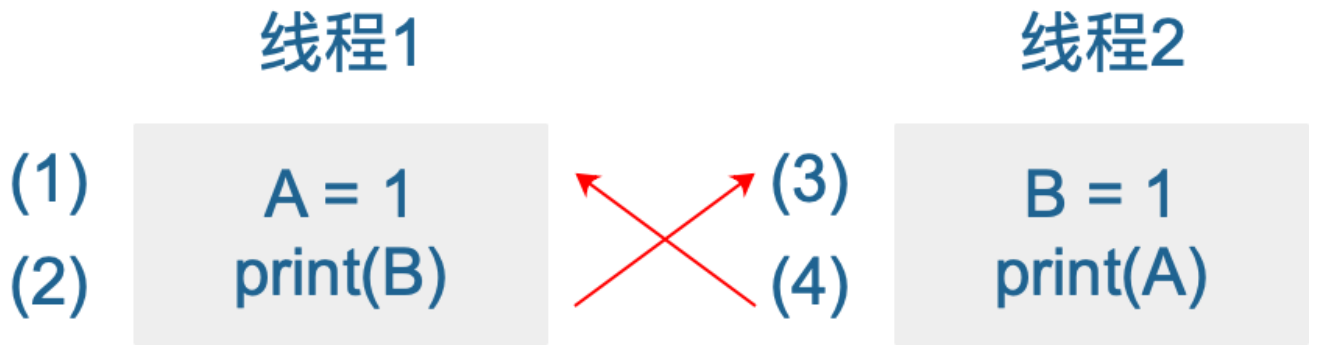
执行顺序	输出结果
1-2-3-4	01
3-4-1-2	01
1-3-2-4	11
1-3-4-2	11

当然，还有一些对称的情形，和上面表格中列出的输出是一样的。例如，执行为顺序为 3-1-4-2 的输出为 11。

从 01 的排列组合来看，总共有4种：00、01、10、11。表格中还差两种：10、00。我们来重点分析下这两种结果究竟会不会出现。

首先是 10，假设 (2) 输出 1，(4) 输出 0。那么首先给 2，3 排个序：(3) -> (2)，因为先要将 B 赋值为 1，(2) 才能打印出 1；同理，(4) -> (1)。另外，因为先打印 1，所以 (2) 要在 (4) 前面，合起来：(3) -> (2) -> (4) -> (1)。(2) 竟然在 (1) 前面执行了，不可能的！

那我们再分析下 00，要想打印 00，打印语句必须在相应变量赋值前执行：



图中箭头表示先后顺序。这就尴尬了，形成了一个环。如果先从 (1) 开始，那顺序就是 (1) -> (2) -> (3) -> (4) -> (1)。(1) 要被执行了 2 次，怎么可能？所以 00 这种情形也是不可能出现的。

但是，上面说的两种情况在真实世界是有可能发生的。曹大的讲义里有验证的方法，感兴起的同学自己去尝试。总共测试了 100 百万次，测试结果如下：

```
Histogram (4 states)
96     *>0:EAX=0; 1:EAX=0; x=1; y=1;
499878:>0:EAX=1; 1:EAX=0; x=1; y=1;
499862:>0:EAX=0; 1:EAX=1; x=1; y=1;
164    :>0:EAX=1; 1:EAX=1; x=1; y=1;
```

非常反直觉，但是在多线程的世界，各种诡异的问题，只有你想不到，没有计算机做不到的。

我们知道，用户写下的代码，先要编译成汇编代码，也就是各种指令，包括读写内存的指令。CPU 的设计者们，为了榨干 CPU 的性能，无所不用其极，各种手段都用上了，你可能听过不少，像流水线、分支预测等等。

其中，为了提高读写内存的效率，会对读写指令进行重新排列，这就是所谓的 内存重排，英文为 Memory Reordering。

这一部分说的是 CPU 重排，其实还有编译器重排。

编译器重排

来看一个代码片段：

```
X = 0
for i in range(100):
    X = 1
    print X
```

这段代码执行的结果是打印 100 个 1。一个聪明的编译器

会分析到循环里对 X 的赋值 `x = 1` 是多余的，每次都要给它赋上 1，完全没必要。因此会把代码优化一下：

```
X = 1
for i in range(100):
    print X
```

优化后的运行结果完全和之前的一样，完美！

但是，如果这时有另外一个线程同时干了这么一件事：

```
X = 0
```

由于这两个线程并行执行，优化前的代码运行的结果可能是这样的：`11101111...`。出现了 1 个 0，但在下次循环中，又会被重新赋值为 1，而且之后一直都是 1。

但是优化后的代码呢：`11100000...`。由于把 `x = 1` 这一条赋值语句给优化掉了，某个时刻 X 变成 `0` 之后，再也没机会变回原来的 `1` 了。

> 在多核心场景下,没有办法轻易地判断两段程序是“等价”的。

可见编译器的重排也是基于运行效率考虑的，但以多线程运行时，就会出各种问题。

为什么要内存重排

引用曹大的一句话：

> 软件或硬件系统可以根据其对代码的分析结果，一定程度上打乱代码的执行顺序，以达到其不可告人的目的。

软件指的是编译器，硬件是 CPU。不可告人的目的就是：

> 减少程序指令数
> 最大化提高 CPU 利用率

曹大又皮了！

内存重排的底层原理

CPU 重排的例子里提到的两种不可能出现的情况，并不是那么显而易见，甚至是难以理解。原因何在？

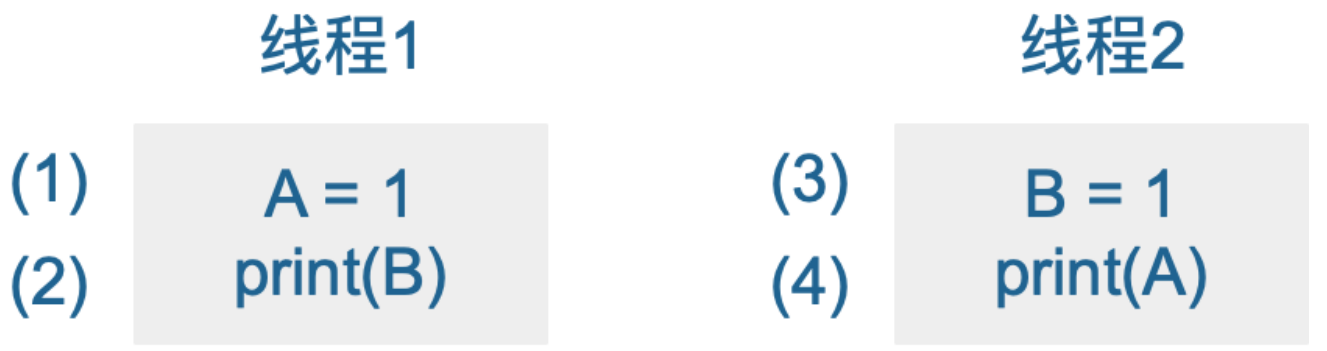
因为我们相信在多线程的程序里，虽然是并行执行，但是访问的是同一块内存，所以没有语句，准确说是指令，能“真正”同时执行的。对同一个内存地址的写，一定是有先有后，先写的结果一定会被后来的操作看到。

当我们写的代码以单线程运行的时候，语句会按我们的本来意图 顺序 地去执行。一旦单线程变成多线程，情况就变了。

想像一个场景，有两个线程在运行，操作系统会在它们之间进行调度。每个线程在运行的时候，都会顺序地执行它的代码。由于对同一个变量的读写，会访问内存的同一地址，所以同一时刻只能有一个线程在运行，即使 CPU 有多个核心：前一个指令操作的结果要让后一个指令看到。

这样带来的后果就是效率低下。两个线程没法做到并行，因为一个线程所做的修改会影响到另一个线程，那后者只能在前者的修改所造成的影响“可见”了之后，才能运行，变成了串行。

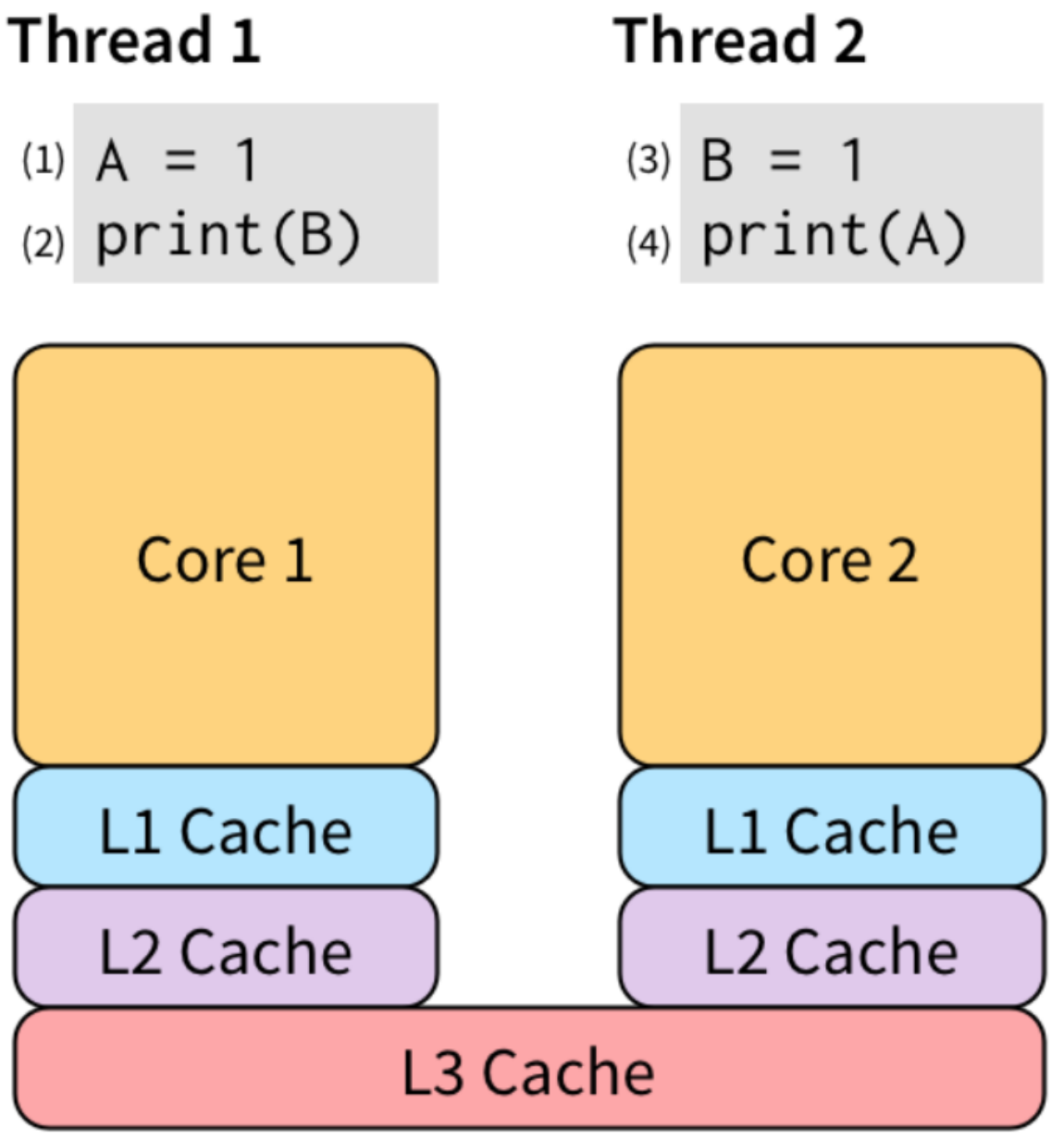
重新来思考前面的例子：



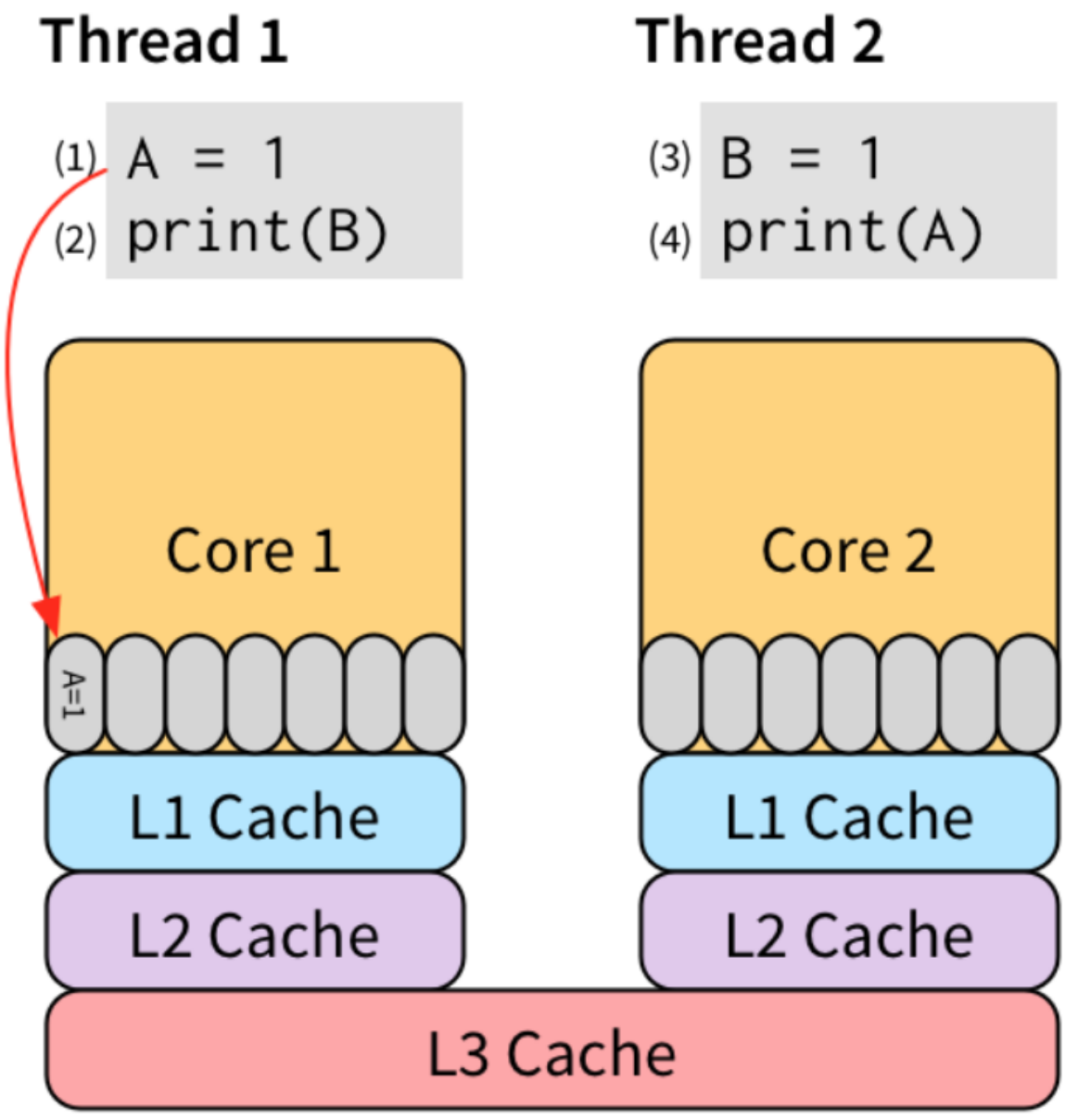
考虑一个问题，为什么 (2) 要等待 (1) 执行完之后才能执行呢？它们之间又没有什么联系，影响不到彼此，完全可以并行去做啊！

由于 (1) 是写语句，所以比 (2) 更耗时，从 `a single view of memory` 这个视角来看，(2) 应该等 (1) 的“效果”对其他所有线程可见了之后才可以执行。但是，在一个现代 CPU 里，这需要花费上百个 CPU 周期。

现代 CPU 为了“抚平”内核、内存、硬盘之间的速度差异，搞出了各种策略，例如三级缓存等。

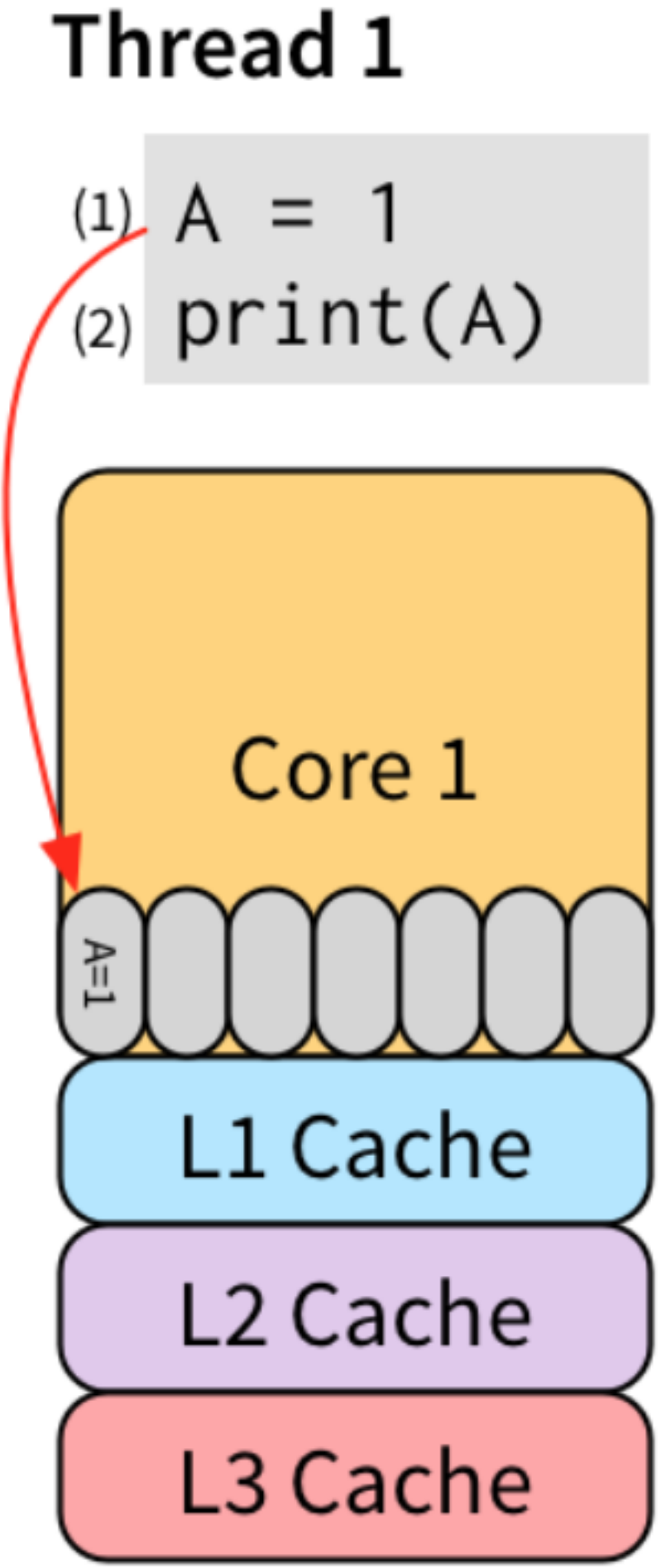


为了让 (2) 不必等待 (1) 的执行“效果”可见之后才能执行，我们可以把 (1) 的效果保存到 `store buffer`：



当 (1) 的“效果”写到了 `store buffer` 后，(2) 就可以开始执行了，不必等到 `A = 1` 到达 L3 cache。因为 `store buffer` 是在内核里完成的，所以速度非常快。在这之后的某个时刻，`A = 1` 会被逐级写到 L3 cache，从而被其他所有线程看到。`store buffer` 相当于把写的耗时隐藏了起来。

`store buffer` 对单线程是完美的，例如：

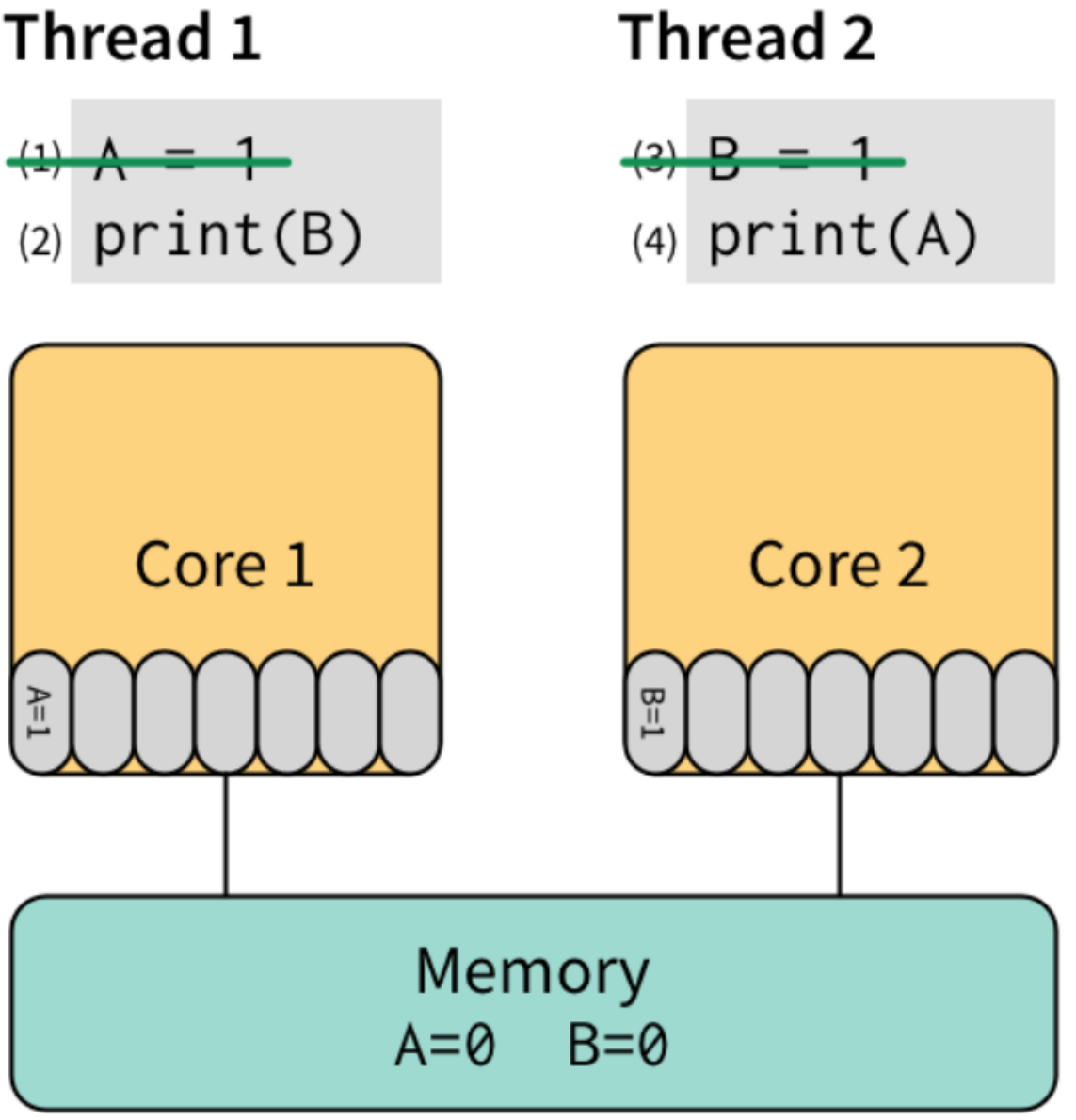


将 (1) 存入 `store buffer` 后，(2) 开始执行。注意，由于

是同一个线程，所以语句的执行顺序还是要保持的。

(2) 直接从 `store buffer` 里读出了 `A = 1`，不必从 L3 Cache 或者内存读取，简直完美！

有了 `store buffer` 的概念，我们再来研究前面的那个例子：



先执行 (1) 和 (3)，将他们直接写入 `store buffer`，接着执行 (2) 和 (4)。“奇迹”要发生了：(2) 看了下 `store buffer`，并没有发现有 B 的值，于是从 Memory 读出了 0，(4) 同样从 Memory 读出了 0。最后，打印出了 `00`。

所有的现代 CPU 都支持 `store buffer`，这导致了很多对程序员来说是难以理解的现象。从某种角度来说，不等 `A = 1` 扩散到 Memory，就去执行 `print(B)` 语句，可以看成读写指令重排。有些 CPU 甚至优化得更多，几乎所有的操作都可以重排，简直是噩梦。

因此，对于多线程的程序，所有的 CPU 都会提供“锁”支持，称之为 `barrier`，或者 `fence`。它要求：

```
A barrier instruction forces all memory operations before it to complete before any memory operation after it can begin.
```

barrier 指令要求所有对内存的操作都必须要“扩散”到 memory 之后才能继续执行其他对 memory 的操作。

barrier 指令要耗费几百个 CPU 周期，而且容易出错。因此，我们可以用高级点的 `atomic compare-and-swap`，或者直接用更高级的锁，通常是标准库提供。

> 正是 CPU 提供的 barrier 指令，我们才能实现应用层的各种同步原语，如 atomic，而 atomic 又是各种更上层的 lock 的基础。

以上说的是 CPU 重排的原理。编译器重排主要是依据语言自己的“内存模型”，不深入了。

出现前面描述的诡异现象的根源在于程序存在 `data race`，也就是说多个线程会同时访问内存的同一个地方，并且至少有一个是写，而且导致了内存重排。所以，最重

要的是当我们在写并发程序的时候，要使用一些“同步”的标准库，简单理解就是各种锁，来避免由于内存重排而带来的一些不可预知的结果。

总结

内存重排是指程序在实际运行时对内存的访问顺序和代码编写时的顺序不一致，主要是为了提高运行效率。分别是硬件层面的 CPU 重排 和软件层面的 编译器重排。

单线程的程序一般不会有太大问题；多线程情况下，有时会出现诡异的现象，解决办法就是使用标准库里的锁。锁会带来性能问题，为了降低影响，锁应该尽量减小粒度，并且不要在互斥区（锁住的代码）放入耗时长的操作。

> lock contention 的本质问题是需要进入互斥区的 goroutine 需要等待独占 goroutine 退出后才能进入互斥区，并行 → 串行。

本文讲的是曹大讲座的一部分，我没有深入研究其他内容，例如 MESI协议、cache contention 等，讲清这些又要牵扯到很多，我还是聚集到深度解密 Go 语言系列吧。有兴趣的话，去曹大博客，给我们提供了很多参考链接，可以自行探索。

参考资料

【曹大 github】https://github.com/cch123/golang-notes/blob/master/memory_barrier.md

【曹大讲义】https://cch123.github.io/ooo/

【内存一致模型】https://homes.cs.washington.edu/~bornholt/post/memory-models.html

【掘金咔叽咔叽，译】https://juejin.im/post/5d0519e05188257a78764d5d#comment