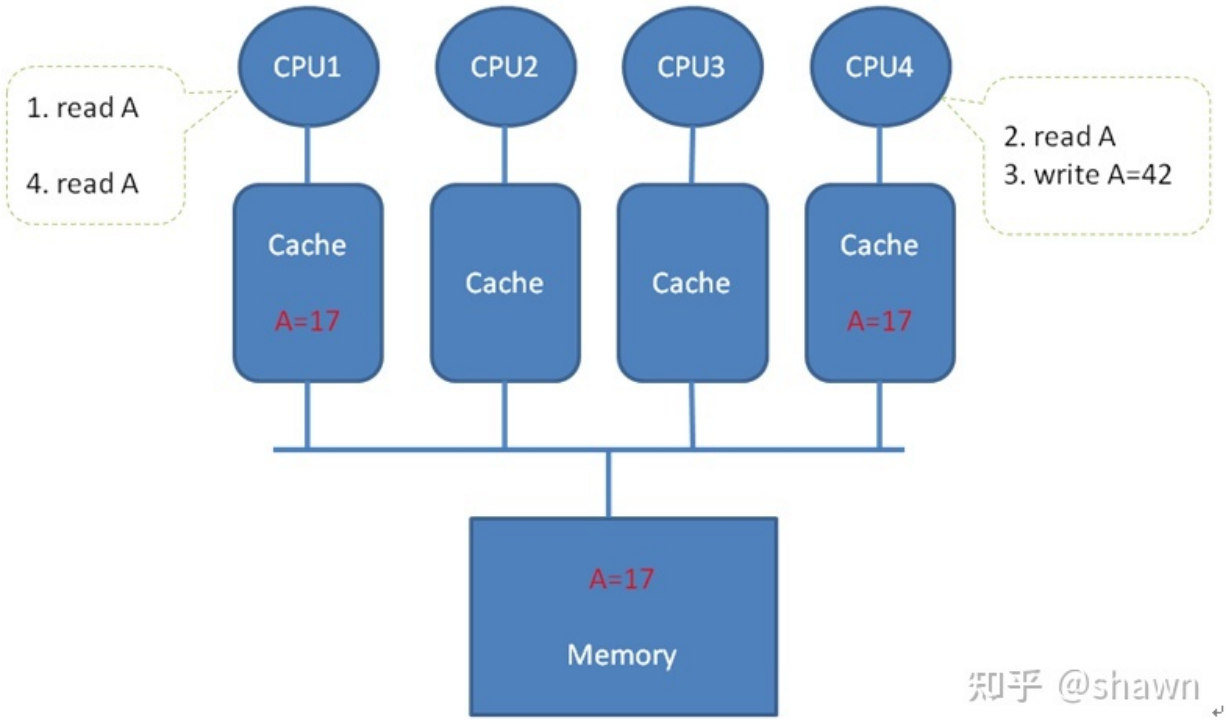
Cache一致性的那些事儿 (1)--什么是cache一致性

1. 什么是Cache一致性?

计算机系统中Cache的引入，降低了内存访问时延，也大幅提升内存访问带宽，在目前的内存技术条件下，以比较经济的手段大幅度地提升了系统的整体性能。 但正如硬币的两个面， cache的引入也带来技术上的复杂性，特别是对于多核系统来讲，共享数据（shared data）的cache一致性问题就是一个比较复杂的课题。

什么是Cache一致性问题呢？我们来看一个例子

在当代的多核计算机体系结构中，当核数不是特别多时，一般每个CPU核都会使用一个本地L1和L2 Cache（称之为private cache），以提升CPU核的整体性能，在多核之间共享一个L3 cache (也叫LLC: last level Cache)和一个全局的主存储器（比如DDR3或DDR4）。



我们假设在主存储器(memory)中有一个全局变量A=17（罗翔老师喜欢用张三，我喜欢用小A）。

Step 1: CPU1先试图读取该全局变量，由于其cache中当前并没有缓存该变量，故发生cache miss, 于是从主内存中加载该变量进入其cache. 结果表现为在其Cache中有一个A=17的本地副本。

Step 2: 随后CPU4也试图读取该全局变量，同样发生cache miss,也会从主内存中加载该变量进入cpu4的cache. 所以在cpu4的Cache中有一个A=17的本地副本。

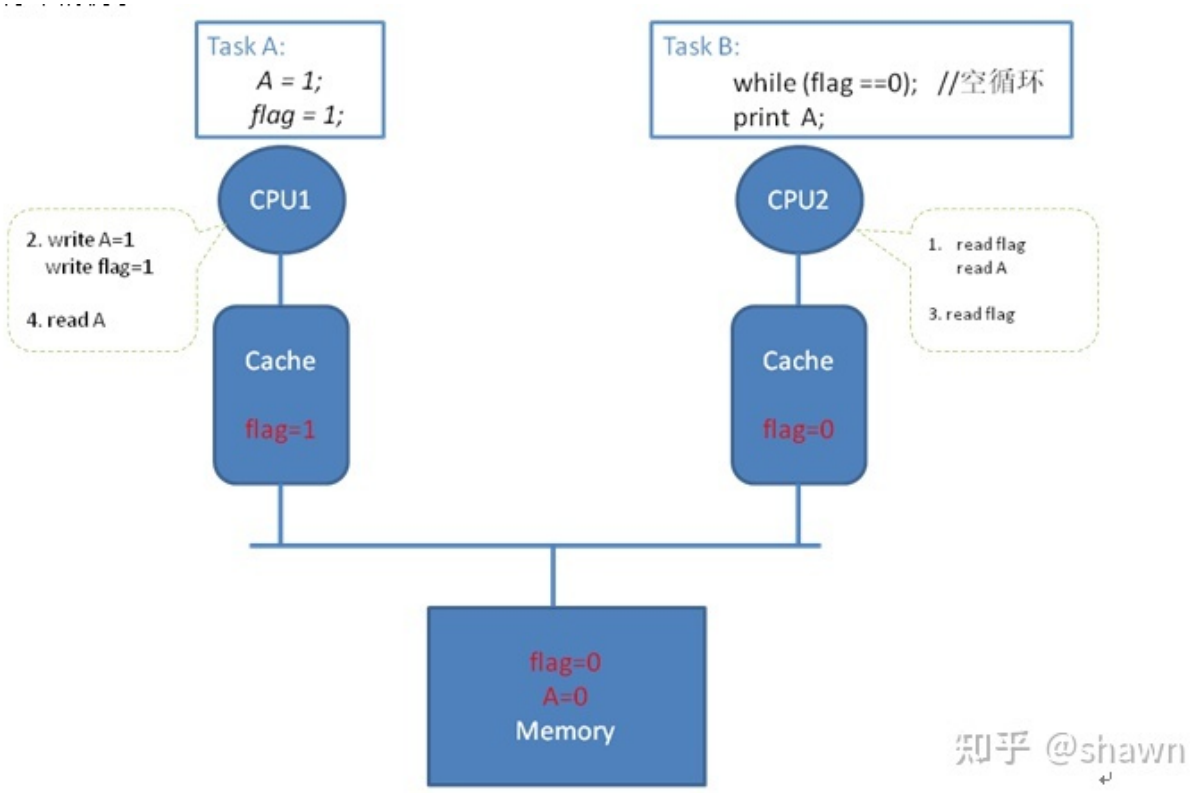
Step 3: CPU4通过计算，需要修改该变量的值，比如设置A=42。此时如果CPU4采用的是write through策略，则A=42会同时反映到CPU4的cache中和主memory中；如果CPU4采用的是write back策略，则A=42

只会反映到CPU4的cache中（设置一个Dirty位），而主memeory继续保持A=17。

Step 4: 此时CPU1继续使用变量A，但由于CPU1的cache中已经缓存了该变量，所以无论step3中CPU4采用的是write back还是wirte through, CPU1看到的只是A=17。

这就产生了Cache的一致性问题，同一个变量在不同的CPU看到的值不一样。

Cache一致性问题导致的后果那是相当严重的，程序无法按照预想进行运行。我们看一个实际中的例子：



我们想充分利用多核系统的并行处理能力；我们在一个多核系统上编写一个多任务的应用，一个进程A(task A)通过设置参数控制另外一个进程(task B)的运行。

首选我们在主存储器(memory)中创建了两个全局变量A=0，flag = 0。比如flag是我们的控制变量，A是我们期望观察到的信号；操作系统把Task A调动在CPU1上运行，把 Task B调度在CPU2上运行。

Step 1: CPU2先试图读取全局变量flag和A，由于其cache中当前并没有缓存该两个变量，故都发生cache miss, 于是从主内存中加载变量flag和变量A进入其cache. 此时CPU2看到的flag==0, 所以保持空循环运行。

Step 2: 随后CPU1试图修改这两个全局变量，同样也发生cache miss,也会先从主内存中加载该变量进入cpu4的cache， 然后对这两个变量的值进行修改，设置A=1，flag=1。

如果采用write through策略，在A=1和flag=1会更新到memory中。如果采用write back则A=1和flag=1仅在CPU1处取作用。

Step 3: CPU2在一次读取flag, 但由于其cache中有一条有效的记录存放着flag==0，所以CPU2此时看到的还是老旧的过时的flag的值，CPU2一直保持在空循环状态，无法打印处A的值。

我们看到了cache一致性（准确地说有coherence和consistence）的重要性，那怎么解决这种cache的不一致性呢？目前有两种主要的解决方案。总线snooping-based方案和Directory-based方案。

Snoopying-based方案：每个cache block对应着看一个共享的状态，系统中所有的cache控制器通过共享总线进行互联，每个cache控制器都监控着共享总线上的操作，根据共享总线上的命令来更新自己的共享状态。

Directory-based方案：在某个单一的位置（directory）对某个cache line的共享状态进行跟踪。