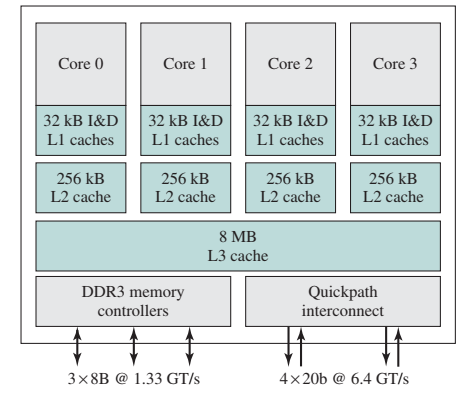
**软件锁与内存一致性模型**

**一、存储层次**





**二、一致性模型**

缓存一致性（Cache Coherence）：How do other CPUs see a memory update?（读操作返回什么样的值）

内存一致性（Memory Consistency）：When does another CPU see a memory update?（写入数据何时才能被读取）

**2.1 缓存一致性问题**

多个线程并发访问同一个共享变量的时候，这些线程的执行处理器（核）上的高速缓存各自都会保留一份该共享变量的副本。由此引发一个问题：一个处理器对其副本数据进行更新之后，其他处理器如何 “感知”到该更新并做出适当反应，以确保这些处理器后续读取该共享变地址时能够读取到这个更新。这就是缓存一致性问题。



实质就是如何防止脏读，确保所有处理器看到的存储器视图是一致的。

硬件实现：MESI（Modified-Exclusive-Shared-Invalid）协议

CPU辅助指令：lock（总线锁、缓存锁）

**2.2 内存一致性问题**

高速缓存一致性确保了多个处理器所看到的存储器视图的一致性，但它并未说明存储器视图必须是怎样的一致性，即被一个处理器修改过的值，在何时必须被其他处理器看到。

处理器之间需要通过共享变量进行通信（包括同步类型的共享变量和资源类型的共享变量），内存一致性问题可归结为：不同处理器对不同单元进行读和写操作时，应该加上何种属性才能确保一致性。

处理器领域还有很多的优化手段，流水线执行、乱序执行、预测执行等等，这些优化会对指令的执行顺序产生很大影响。不同处理器实现指令重排的方式不尽相同（load-load、load-store、store-store、store-load等等），对于程序员来说，不应该受限于某一个处理器的实现细节，来保证程序的正确与否。

**内存严格一致性模型（Strict Consistency）**：共享变量的读写完全序列化



当处理器P2第一次读取x的值时，它并没读到处理器P1在之前对x所写入的值1，但是它最终还是读取到了他的值。这种模型称为“内存原子一致性”(atomic consistency)

**顺序一致性内存模型（Sequential Consistency）：**“（并发程序在多处理器上的）任何一次执行结果都相同，就像所有处理器的操作按照某个顺序执行，各个微处理器的操作按照其程序指定的顺序进行。”（Lamport）



**缓存一致性（Cache Coherence）：**见2.1

**数据竞争：**一个变量被多个进程（线程）修改，而结果决定与哪个进程（线程）先执行。为了让这个程序变得可靠，必须使用锁来保证，其中的一个进程（线程）在另一个进程（线程）开始之前先执行完整个操作过程。



4种不同的执行顺序，会导致3中不同的结果：（变量修改——load、Operation、store三个步骤）

* P1先执行 --- x获得的值会是3。
* P2先执行 --- x获得的值会是3。
* P1和P2都读取了x的值，P1先执行对x的写操作 --- x获得的值会是2。
* P1和P2都读取了x的值，P2先执行对x的写操作 --- x获得的值会是1。

**编译优化**：

大多数的语言支持一种语义，在这种语义中程序的执行顺序是通过每一个内存地址来维护的，但是不能跨内存地址；这就给了编译器重排代码顺序的自由。因此，比如，如果一个程序写两个变量x和y，这两个变量相互之间没有依赖关系，那么编译器就有自由（权利）去以任意的顺序来执行对这两个变量的写操作而不影响程序的正确性。然而，在一个并行的环境中，很可能一个在其它一些处理器上运行的程序确实依赖于x和y变量的写操作发生的顺序。

两个互相排斥的进程是一个很好的例子。进入临界区的代码如下所示：



如果编译器决定（不管以什么理由）交换对flag[i]和trun的写操作的顺序，这在单线程环境中是完全正确的代码，但是在多线程环境中却会导致失败。

处理器支持乱序执行，不会保证程序的机器码将会按照指定的顺序来执行对内存的访问；

处理器和缓存的紧密耦合，所以处理器会进行更加激进的指令重排，这种类型的优化几乎没有办法阻止或者进行控制；

（例如，处理器已经完成了对turn的更新，但是它仍然在设置上面的flag[i]的值，因为访问flag[i]涉及到了对数组的访问）。

**同步内存访问**

一种正确编写具有共享内存变量的并行程序的方法是使用互斥的方法来保护要访问的共享内存变量。在上面第一个有bug的例子中，通过加锁能够得到确定性的行为代码，用S表示相关的同步操作。



一般来说，在正确的并行程序中，我们先获得对一系列共享变量的互斥的访问，然后就可以按照预想的要求来进行处理，之后退出互斥访问，向系统其它的部分分发修改之后的共享变量的值。其它的处理器没有必要看到值得中间结果；它们仅仅只要看到最终的结果就可以了。

内存访问模型的分类图：



**解决实验二的方法：**

内存屏障（读、写、读写）