2.4 多线程编程的原则及要点：

随着多核 CPU 的出世，多核编程方面的问题将摆上了程序员的日程，有许多老的程序员以为早就有多 CPU 的机器，业界在多 CPU 机器上的编程已经积累了很多经验，多核 CPU 上的编程应该差不多，只要借鉴以前的多任务编程、并行编程和并行算法方面的经验就足够了。

但是，多核机器和以前的多 CPU 机器有很大的不同，以前的多 CPU 机器都是用在特定领域，比如服务器，或者一些可以进行大型并行计算的领域，这些领域很容易发挥出多 CPU 的优势，而现在多核机器则是应用到普通用户的各个层面，特别是客户端机器要使用多核 CPU，而很多客户端软件要想发挥出多核的并行优势恐怕没有服务器和可以进行大型并行计算的特定领域简单。

多核 CPU 中，要很好地发挥出多个 CPU 的性能的话，必须保证分配到各个 CPU 上的任务有一个很好的负载平衡。否则一些 CPU 在运行，另外一些 CPU 处于空闲，无法发挥出多核 CPU 的优势来。

要实现一个好的负载平衡通常有两种方案，一种是静态负载平衡，另外一种是动态负载平衡。

1、静态负载平衡
静态负载平衡中，需要人工将程序分割成多个可并行执行的部分，并且要保证分割成的各个部分能够均衡地分布到各个 CPU 上运行，也就是说工作量要在多个任务间进行均匀的分配，使得达到高的加速系数。

2、动态负载平衡
动态负载平衡是在程序的运行过程中来进行任务的分配达到负载平衡的目的。实际情况中存在许多不能由静态负载平衡解决的问题，比如一个大的循环中，循环的次数是由外部输入的，事先并不知道循环的次数，此时采用静态负载平衡划分策略就很难实现负载平衡。

动态负载平衡中对任务的调度一般是由系统来实现的，程序员通常只能选择动态平衡的调度策略，不能修改调度策略，由于实际任务中存在很多的不确定因素，调度算法无法做得很优，因此动态负载平衡有时可能达不到既定的负载平衡要求。

3、负载平衡的难题在那里？
负载平衡的难题并不在于负载平衡的程度要达到多少，因为即使在各个 CPU 上分配的任务执行时间存在一些差距，但是随着 CPU 核数的增多总能让总的执行时间下降，从而使加速系数随 CPU 核数的增加而增加。

负载平衡的困难之处在于程序中的可并行执行块很多要靠程序员来划分，当然 CPU 核数较少时，比如双核或 4 核，这种划分并不是很困难。但随着核数的增加，划分的粒度将变得越来越细，到了 16 核以上时，估计程序员要为如何划分任务而抓狂。比如一段顺序执行的代码，放到 128 核的 CPU 上运行，要手工划分成 128 个任务，其划分的难度可想而知。

负载划分的误差会随着 CPU 核数的增加而放大，比如一个需要 16 个时间单位的程序分到 4 个任务上执行，平均每个任务上的负载执行时间为 4 个时间单位，划分误差为 1 个时间单位的话，那么加速系数变成 16/(4+1)=3.2，是理想情况下加速系数 4 的 80%。但是如果放到一个 16 核 CPU 上运行的话，如果某个任务的划分误差如果为 0.5 个时间单位的话，那么加速系数变成 16/(1+0.5) = 10.67，只有理想的加速系数 16 的 66.7%，如果核数再增加的话，由于误差的放大，加速系数相比于理想加速系数的比例还会下降。

负载划分的难题还体现在 CPU 和软件的升级上，比如在 4 核 CPU 上的负载划分是均衡的，但到了 8 核、16 核上，负载也许又变得不均衡了。软件升级也一样，当软件增加功能后，负载平衡又会遭到破坏，又需要重新划分负载使其达到平衡，这样一来软件设计的难度和麻烦大大增加了。

难题一：串行化方面的难题
1）加速系数
衡量多处理器系统的性能时，通常要用到的一个指标叫做加速系数，定义如下：
S(p) = 使用单处理器执行时间（最好的顺序算法）/ 使用具有 p 个处理器所需执行时间

2）Amdahl 定律
并行处理时有一个 Amdahl 定律，用方程式表示如下：
S(p) = p / (1 + (p-1)*f)
其中 S(p)表示加速系数
p 表示处理器的个数
f 表示串行部分所占整个程序执行时间的比例
当 f = 5%，p = 20 时，S(p) = 10.256 左右
当 f = 5%，p = 100 时，S(p) = 16.8 左右
也就是说只要有 5%的串行部分，当处理器个数从 20 个增加到 100 个时，加速系数只能从 10.256 增加到 16.8 左右，处理器个数增加了 5 倍，速度只增加了 60%多一点。即使处理器个数增加到无穷多个，加速系数的极限值也只有 20。

如果按照 Amdahl 定律的话，可以说多核方面几乎没有任何发展前景，即使软件中只有 1%的不可并行化部分，那么最大加速系统也只能到达 100，再多的 CPU 也无法提升速度性能。按照这个定律，可以说多核 CPU 的发展让摩尔定律延续不了多少年就会到达极限。

3）Gustafson 定律
Gustafson 提出了和 Amdahl 定律不同的假设来证明加速系数是可以超越 Amdahl 定律的限制的，Gustafson 认为软件中的串行部分是固定的，不会随规模的增大而增大，并假设并行处理部分的执行时间是固定的（服务器软件可能就是这样）。Gustafson 定律用公式描述如下：

S(p) = p + (1-p)*fts
其中 fts 表示串行执行所占的比例

如果串行比例为 5%，处理器个数为 20 个，那么加速系数为 20＋(1-20)*5%=19.05

如果串行比例为 5%，处理器个数为 100 个，那么加速系数为 100＋(1-100)*5%=95.05

Gustafson 定律中的加速系数几乎跟处理器个数成正比，如果现实情况符合 Gustafson 定律的假设前提的话，那么软件的性能将可以随着处理个数的增加而增加。

4）实际情况中的并行化分析

Amdahl 定律和 Gustafson 定律的计算结果差距如此之大，那么现实情况到底是符合那一个定律呢？我个人认为现实情况中既不会象 Amdahl 定律那么悲观，但也不会象 Gustafson 定律那么乐观。为什么这样说呢？还是进行一下简单的分析吧。

首先需要确定软件中到底有那么内容不能并行化，才能估计出串行部分所占的比例，20 世纪 60 年代时，Bernstein 就给出了不能进行并行计算的三个条件：

条件 1：C1 写某一存储单元后，C2 读该单元的数据。称为“写后读”竞争

条件 2：C1 读某一存储单元数据后，C2 写该单元。称为“读后写”竞争

条件 3：C1 写某一存储单元后，C2 写该单元。称为“写后写”竞争

满足以上三个条件中的任何一个都不能进行并行执行。不幸的是在实际的软件中大量存在满足上述情况的现象，也就是我们常说的共享数据要加锁保护的问题。

加锁保护导致的串行化问题如果在任务数量固定的前提下，串行化所占的比例是随软件规模的增大而减小的，但不幸的是它会随任务数量的增加而增加，也就是说处理器个数越多，锁竞争导致的串行化将越严重，从而使得串行化所占的比例随处理器个数的增加而急剧增加。所以串行化问题是多核编程面临的一大难题。

5）可能的解决措施

对于串行化方面的难题，首先想到的解决措施就是少用锁，甚至采用无锁编程，不过这对普通程序员来说几乎是难以完成的工作，因为无锁编程方面的算法太过于复杂，而且使用不当很容易出错，许多已经发表到专业期刊上的无锁算法后来又被证明是错的，可以想象得到这里面的难度有多大。

第二个解决方案就是使用原子操作来替代锁，使用原子操作本质上并没有解决串行化问题，只不过是让串行化的速度大大提升，从而使得串行化所占执行时间比例大大下降。不过目前芯片厂商提供的原子操作很有限，只能在少数地方起作用，芯片厂商在这方面可能还需要继续努力，提供更多功能稍微强大一些的原子操作来避免更多的地方的锁的使用。

第三个解决方案是从设计和算法层面来缩小串行化所占的比例。也许需要发现实用的并行方面的设计模式来缩减锁的使用，目前业界在这方面已经积累了一定的经验，如任务分解模式，数据分解模式，数据共享模式，相信随着多核 CPU 的大规模使用将来会有更多的新的有效的并行设计模式和算法冒出来。

第四个解决方案是从芯片设计方面来考虑的。主要的想法是在芯片层面设计一些新的指令，这些指令不象以前单核 CPU 指令那样是由单个 CPU 完成的，而是由多个 CPU 进行并行处理完成的一些并行指令，这样程序员调用这些并行处理指令编程就象编写串行化程序一样，但又充分利用上了多核的优势。

前面我们提到了锁竞争会让程序中的串行化比例随使用的 CPU 的核数增多而加剧的现象，现在我们就来对多核编程中的锁竞争进行深入的分析。

为了简化起见，我们先看一个简单的情况，假设有 4 个对等的任务同时启动运行，假设每个任务刚开始时有一个需要锁保护的操作，耗时为 1，每个任务其他部分的耗时为 25。这几个任务启动运行后的运行情况如下图所示：

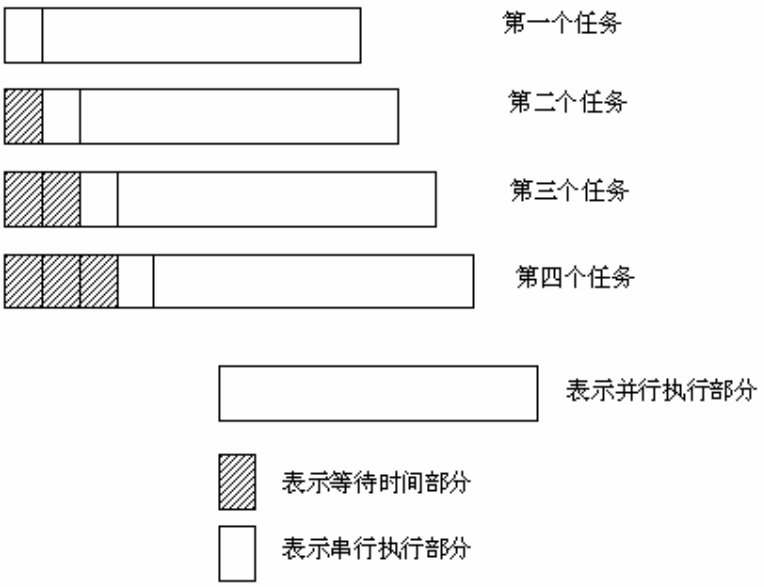

图 2.9：对等任务的锁竞争示意图

在上图中，可以看出第 1 个任务直接执行到结束，中间没有等待，第 2 个任务等待了 1 个时间单位，第 3 个任务等待了 2 个时间单位，第 3 个任务等待了 3 个时间单位。

这样整个任务总计等待了 6 个时间单位，如果这几个任务是采用 OpenMP 里的所有任务都在同一点上进行等待到全部任务执行完再向下执行时，那么总的运行时间将和第四个任务一样为 29 个时间单位，加速系数为：（1＋4×25）/ 29 ＝ 3.48

即使以 4 个任务的平均时间 27.5 来进行计算，加速系数＝101/27.5 = 3.67

按照 Amdahl 定律来计算加速系数的话，上述应用中，串行时间为 1，并行处理的总时间转化为串行后为 100 个时间单位，如果放在 4 核 CPU 上运行的话，加速系数＝p / (1 + (p-1)*f) ＝ 4/(1+(4-1)*1/101) = 404/104 = 3.88

这就产生了一个奇怪的问题，使用了锁之后，加速系数连阿姆尔达定律计算出来的加速系数都不如，更别说用 Gustafson 定律计算的加速系数了。

其实可以将上面 4 个任务的锁竞争情况推广到更一般的情况，假设有锁保护的串行化时间为 1，可并行化部分在单核 CPU 上的运行时间为 t，CPU 核数为 p，那么在 p 个对等任务同时运行情况下，锁竞争导致的总等待时间为：1＋2＋…＋p ＝ p*(p-1)/2

耗时最多的一个任务所用时间为： (p-1) + t/p

使用耗时最多的一个任务所用时间来当作并行运行时间的话，加速系数如下

S(p) = t / (p-1 + t/p) = p*t / (p*(p-1)+t)　　（锁竞争下的加速系数公式）

这个公式表明在有锁竞争情况下，如果核数固定情况下，可并行化部分越大，那么加速系数将越大。在并行化时间固定的情况下，如果 CPU 核数越多，那么加速系数将越小。

还是计算几个实际的例子来说明上面公式的效果：
令 t=100，p=4，加速系数＝4×100 / (4*(4-1)+100) = 3.57
令 t=100，p=16，加速系数＝16×100 / (16*(16-1)+100) = 4.7
令 t=100，p=64，加速系数＝64×100 / (64*(64-1)+100) = 1.54
令 t=100，p=128，加速系数＝128×100 / (128*(128-1)+100) = 0.78

从以上计算可以看出，当核数多到一定的时候，加速系数不仅不增加反而下降，核数增加到 128 时，加速系数只有 0.78，还不如在单核 CPU 上运行的速度。

上面的例子中，锁保护导致的串行代码是在任务启动时调用的，其实对等任务中在其他地方调用的锁保护的串行代码也是一样的。

对等型任务的锁竞争现象在实际情况中是很常见的，比如服务器软件，通常各个客户端处理任务都是对等的，如果在里面使用了锁的话，那么很容易造成上面说的加速系数随 CPU 核数增多而下降的现象。

以前的服务器软件一般运行在双 CPU 或四 CPU 机器上，所以锁竞争导致的加速系数下降现象不明显，进入多核时代后，随着 CPU 核数的增多，这个问题将变得很严重，所以多核时代对程序设计提出了新的挑战。以前的多任务下的编程思想放到多核编程上不一定行得通。

所以简单地认为多核编程和以前的多任务编程或并行计算等同的话是不切实际的。

当然由于目前市面上销售的多核 CPU 还是双核和四核的，等到 16 核以上的 CPU 大规模进入市场可能还有几年时间，相信业界在未来的几年内能够对于上面对等任务上的锁竞争问题找到更好的解决方案。