8.2 如何让数据紧凑？

`多处理系统中，使用并发的方式来提高代码的效率时，你需要了解一下有哪些因素会影响并发的效率。即使已经使用多线程对关注进行分离，还需要确定是否会对性能造成负面影响。因为，在16核机器上应用的速度与单核机器相当时，用户是不会打死你的。

之后你会看到，在多线程代码中有很多因素会影响性能——对线程处理的数据做一些简单的改动(其他不变)，都可能对性能产生戏剧性的效果。所以，多言无益，让我们来看一下这些因素吧，从明显的开始：目标系统有多少个处理器？

8.2.1 有多少个处理器？

处理器个数是影响多线程应用的首要因素。在某些情况下，你对目标硬件会很熟悉，并且针对硬件进行设计，并在目标系统或副本上进行测量。如果是这样，那你很幸运；不过，要知道这些都是很奢侈的。你可能在一个类似的平台上进行开发，不过你所使用的平台与目标平台的差异很大。例如，你可能会在一个双芯或四芯的系统上做开发，不过你的用户系统可能就只有一个处理器(可能有很多芯)，或多个单芯处理器，亦或是多核多芯的处理器。在不同的平台上，并发程序的行为和性能特点就可能完全不同，所以你需要仔细考虑那些地方会被影响到，如果会被影响，就需要在不同平台上进行测试。

一个单核16芯的处理器和四核双芯或十六核单芯的处理器相同：在任何系统上，都能运行16个并发线程。当线程数量少于16个时，会有处理器处于空闲状态(除非系统同时需要运行其他应用，不过我们暂时忽略这种可能性)。另一方面，当多于16个线程在运行的时候(都没有阻塞或等待)，应用将会浪费处理器的运算时间在线程间进行切换，如第1章所述。这种情况发生时，我们称其为*超额认购*(oversubscription)。

为了扩展应用线程的数量，与硬件所支持的并发线程数量一致， C++ 标准线程库提供了 `std::thread::hardware_concurrency()` 。使用这个函数就能知道在给定硬件上可以扩展的线程数量了。

需要谨慎使用 `std::thread::hardware_concurrency()` ，因为代码不会考虑有其他运行在系统上的线程(除非已经将系统信息进行共享)。最坏的情况就是，多线程同时调用 `std::thread::hardware_concurrency()` 函数来对线程数量进行扩展，这样将导致庞大的超额认

购。 `std::async()` 就能避免这个问题，因为标准库会对所有的调用进行适当的安排。同样，谨慎的使用线程池也可以避免这个问题。

不过，即使你已经考虑到所有在应用中运行的线程，程序还要被同时运行的其他程序所影响。虽然，在单用户系统中，使用多个CPU密集型应用程序很罕见，但在某些领域，这种情况就很常见了。虽然系统能提供选择线程数量的机制，但这种机制已经超出 C++ 标准的范围。这里的一种选择是使用与 `std::async()` 类似的工具，来为所有执行异步任务的线程的数量做考虑；另一种选择就是，限制每个应用使用的处理芯个数。我倒是希望，这种限制能反映到 `std::thread::hardware_concurrency()` 上面(不能保证)。如果你需要处理这种情况，可以看一下你所使用的系统说明，了解一下是否有相关选项可供使用。

理想算法可能会取决于问题规模与处理单元的比值。大规模并行系统中有很多的处理单元，算法可能就会同时执行很多操作，让应用更快的结束；这就要快于执行较少操作的平台，因为该平台上的每一个处理器只能执行很少的操作。

随着处理器数量的增加，另一个问题就会来影响性能：多个处理器尝试访问同一个数据。

8.2.2 数据争用与乒乓缓存

当两个线程并发的在不同处理器上执行，并且对同一数据进行读取，通常不会出现问题；因为数据将会拷贝到每个线程的缓存中，并且可以让两个处理器同时进行处理。不过，当有线程对数据进行修改的时候，这个修改需要更新到其他核芯的缓存中去，就要耗费一定的时间。根据线程的操作性质，以及使用到的内存序，这样的修改可能会让第二个处理器停下来，等待硬件内存更新缓存中的数据。即便是精确的时间取决于硬件的物理结构，不过根据CPU指令，这是一个特别特别慢的操作，相当于执行成百上千个独立指令。

思考下面简短的代码段：

```
std::atomic<unsigned long> counter(0);
void processing_loop()
{
  while(counter.fetch_add(1,std::memory_order_relaxed)<100000000)
  {
    do_something();
  }
}
```

counter变量是全局的，所以任何线程都能调用processing_loop()去修改同一个变量。因此，当新增加的处理器时，counter变量必须要在缓存内做一份拷贝，再改变自己的值，或其他线程以发布的方式对缓存中的拷贝副本进行更新。即使用 `std::memory_order_relaxed` ，编译器不会为任何数据做同步操作，fetch_add是一个“读-改-写”操作，因此就要对最新的值进行检索。如果另一个线程在另一个处理器上执行同样的代码，counter的数据需要在两个处理器之间进行传递，那么这两个处理器的缓存中间就存有counter的最新值(当counter的值增加时)。如果do_something()足够短，或有很多处理器来对这段代码进行处理时，处理器将会互相等待；一个处理器准备更新这个值，另一个处理器正在修改这个值，所以该处理器就不得不等待第二个处理器更新完成，并且完成更新传递时，才能执行更新。这种情况被称为*高竞争*(high contention)。如果处理器很少需要互相等待，那么这种情况就是*低竞争*(low contention)。

在这个循环中，counter的数据将在每个缓存中传递若干次。这就叫做*乒乓缓存*(cache ping-pong)，这种情况会对应用的性能有着重大的影响。当一个处理器因为等待缓存转移而停止运行时，这个处理器就不能做任何事情，所以对于整个应用来说，这就是一个坏消息。

你可能会想，这种情况不会发生在你身上；因为，你没有使用任何循环。你确定吗？那么互斥锁呢？如果你需要在循环中放置一个互斥量，那么你的代码就和之前从数据访问的差不多了。为了锁住互斥量，另一个线程必须将数据进行转移，就能弥补处理器的互斥性，并且对数据进行修改。当这个过程完成时，将会再次对互斥量进行修改，并对线程进行解锁，之后互斥数据将会传递到下一个需要互斥量的线程上去。转移时间，就是第二个线程等待第一个线程释放互斥量的时间：

```
std::mutex m;
my_data data;
void processing_loop_with_mutex()
{
  while(true)
  {
    std::lock_guard<std::mutex> lk(m);
    if(done_processing(data)) break;
  }
}
```

接下来看看最糟糕的部分：数据和互斥量已经准备好让多个线程进访问之后，当系统中的核心数和处理器数量增加时，很可能看到高竞争，以及一个处理器等待其他处理器的情况。如果在多线程情况下，能更快的对同样级别的数据进行处理，线程就会对数据和互斥量进行竞争。这里有很多这样的情况，很多线程会同时尝试对互斥量进行获取，或者同时访问变量，等等。

互斥量的竞争通常不同于原子操作的竞争，最简单的原因是，互斥量通常使用操作系统级别的序列化线程，而非处理器级别的。如果有足够的线程去执行任务，当有线程在等待互斥量时，操作

系统会安排其他线程来执行任务，而处理器只会在其他线程运行在目标处理器上时，让该处理器停止工作。不过，对互斥量的竞争，将会影响这些线程的性能；毕竟，只能让一个线程在同一时间运行。

回顾第3章，一个很少更新的数据结构可以被一个“单作者，多读者”互斥量(详见3.3.2)。乒乓缓存效应可以抵消互斥所带来的收益(工作量不利时)，因为所有线程访问数据(即使是读者线程)都会对互斥量进行修改。随着处理器对数据的访问次数增加，对于互斥量的竞争就会增加，并且持有互斥量的缓存行将会在核芯中进行转移，因此会增加不良的锁获取和释放次数。有一些方法可以改善这个问题，其本质就是让互斥量对多行缓存进行保护，不过这样的互斥量需要自己去实现。

如果乒乓缓存是一个糟糕的现象，那么该怎么避免它呢？在本章后面，答案会与提高并发潜能的指导意见相结合：减少两个线程对同一个内存位置的竞争。

虽然，要实现起来并不简单。即使给定内存位置被一个线程所访问，可能还是会有乒乓缓存的存在,是因为另一种叫做*伪共享*(false sharing)的效应。

8.2.3 伪共享

处理器缓存通常不会用来处理在单个存储位置，但其会用来处理称为*缓存行*(cache lines)的内存块。内存块通常大小为32或64字节，实际大小需要由正在使用着的处理器模型来决定。因为硬件缓存进处理缓存行大小的内存块，较小的数据项就在同一内存行的相邻内存位置上。有时，这样的设定还是挺不错：当线程访问的一组数据是在同一数据行中，对于应用的性能来说就要好于向多个缓存行进行传播。不过，当在同一缓存行存储的是无关数据，且需要被不同线程访问，这就会造成性能问题。

假设你有一个int类型的数组，并且有一组线程可以访问数组中的元素，且对数组的访问很频繁(包括更新)。通常int类型的大小要小于一个缓存行，同一个缓存行中可以存储多个数据项。因此，即使每个线程都能对数据中的成员进行访问，硬件缓存还是会产生乒乓缓存。每当线程访问0号数据项，并对其值进行更新时，缓存行的所有权就需要转移给执行该线程的处理器，这仅是为了让更新1号数据项的线程获取1号线程的所有权。缓存行是共享的(即使没有数据存在)，因此使用*伪共享*来称呼这种方式。这个问题的解决办法就是对数据进行构造，让同一线程访问的数据项存在临近的内存中(就像是放在同一缓存行中)，这样那些能被独立线程访问的数据将分布在相距很远的地方，并且可能是存储在不同的缓存行中。在本章接下来的内容中看到，这种思路对代码和数据设计的影响。

如果多线程访问同一内存行是一种糟糕的情况，那么在单线程下的内存布局将会如何带来哪些影响呢？

8.2.4 如何让数据紧凑？

伪共享发生的原因：某个线程所要访问的数据过于接近另一线程的数据，另一个是与数据布局相关的陷阱会直接影响单线程的性能。问题在于数据过于接近：当数据能被单线程访问时，那么数据就已经在内存中展开，就像是分布在不同的缓存行上。另一方面，当内存中有能被单线程访问紧凑的数据时，就如同数据分布在同一缓存行上。因此，当数据已传播，那么将会有更多的缓存行将会从处理器的缓存上加载数据，这会增加访问内存的延迟，以及降低数据的系能(与紧凑的数据存储地址相比较)。

同样的，如果数据已传播，在给定缓存行上就即包含于当前线程有关和无关的数据。在极端情况下，当有更多的数据存在于缓存中，你会对数据投以更多的关注，而非这些数据去做了什么。这就会浪费宝贵的缓存空间，增加处理器缓存缺失的情况，即使这个数据项曾经在缓存中存在过，还需要从主存中添加对应数据项到缓存中，因为在缓存中其位置已经被其他数据所占有。

现在，对于单线程代码来说就很关键了，何至于此呢？原因就是*任务切换*(task switching)。如果系统中的线程数量要比核芯多，每个核上都要运行多个线程。这就会增加缓存的压力，为了避免伪共享，努力让不同线程访问不同缓存行。因此，当处理器切换线程的时候，就要对不同内存行上的数据进行重新加载(当不同线程使用的数据跨越了多个缓存行时)，而非对缓存中的数据保持原样(当线程中的数据都在同一缓存行时)。

如果线程数量多于内核或处理器数量，操作系统可能也会选择将一个线程安排给这个核芯一段时间，之后再安排给另一个核芯一段时间。因此就需要将缓存行从一个内核上，转移到另一个内核上；这样的话，就需要转移很多缓存行，也就意味着要耗费很多时间。虽然，操作系统通常避免这样的情况发生，不过当其发生的时候，对性能就会有很大的影响。

当有超级多的线程准备运行时(非等待状态)，任务切换问题就会频繁发生。这个问题我们之前也接触过：超额认购。

8.2.5 超额认购和频繁的任务切换

多线程系统中，通常线程的数量要多于处理的数量。不过，线程经常会花费时间来等待外部I/O完成，或被互斥量阻塞，或等待条件变量，等等；所以等待不是问题。应用使用额外的线程来完成有用的工作，而非让线程在处理器处以闲置状态时继续等待。

这也并非长久之计，如果有很多额外线程，就会有很多线程准备执行，而且数量远远大于可用处理器的数量，不过操作系统就会忙于在任务间切换，以确保每个任务都有时间运行。如第1章所见，这将增加切换任务的时间开销，和缓存问题造成同一结果。当无限制的产生新线程，超额认

购就会加剧，如第4章的递归快速排序那样；或者在通过任务类型对任务进行划分的时候，线程数量大于处理器数量，这里对性能影响的主要来源是CPU的能力，而非I/O。

如果只是简单的通过数据划分生成多个线程，那可以限定工作线程的数量，如8.1.2节中那样。如果超额认购是对工作的天然划分而产生，那么不同的划分方式对这种问题就没有太多益处了。之前的情况是，需要选择一个合适的划分方案，可能需要对目标平台有着更加详细的了解，不过这也只限于性能已经无法接受，或是某种划分方式已经无法提高性能的时候。

其他因素也会影响多线程代码的性能。即使CPU类型和时钟周期相同，乒乓缓存的开销可以让程序在两个单核处理器和在一个双核处理器上，产生巨大的性能差，不过这只是那些对性能影响可见的因素。接下来，让我们看一下这些因素如何影响代码与数据结构的设计。