# 第 10 章　多处理器调度（高级）

本章将介绍多处理器调度（multiprocessor scheduling）的基础知识。由于本章内容相对较深，建议认真学习并发相关的内容后再读。

过去很多年，多处理器（multiprocessor）系统只存在于高端服务器中。现在，它们越来越多地出现在个人 PC、笔记本电脑甚至移动设备上。多核处理器（multicore）将多个 CPU 核组装在一块芯片上，是这种扩散的根源。由于计算机的架构师们当时难以让单核 CPU 更快，同时又不增加太多功耗，所以这种多核 CPU 很快就变得流行。现在，我们每个人都可以得到一些 CPU，这是好事，对吧？

当然，多核 CPU 带来了许多困难。主要困难是典型的应用程序（例如你写的很多 C 程序）都只使用一个 CPU，增加了更多的 CPU 并没有让这类程序运行得更快。为了解决这个问题，不得不重写这些应用程序，使之能并行（parallel）执行，也许使用多线程（thread，本书的第 2 部分将用较多篇幅讨论）。多线程应用可以将工作分散到多个 CPU 上，因此 CPU 资源越多就运行越快。

**补充：高级章节**

需要阅读本书的更多内容才能真正理解高级章节，但这些内容在逻辑上放在一章里。例如，本章是关于多处理器调度的，如果先学习了中间部分的并发知识，会更有意思。但是，从逻辑上它属于本书中虚拟化（一般）和 CPU 调度（具体）的部分。因此，建议不按顺序学习这些高级章节。对于本章，建议在本书第 2 部分之后学习。

除了应用程序，操作系统遇到的一个新的问题是（不奇怪！）多处理器调度（multiprocessor scheduling）。到目前为止，我们讨论了许多单处理器调度的原则，那么如何将这些想法扩展到多处理器上呢？还有什么新的问题需要解决？因此，我们的问题如下。

**关键问题：如何在多处理器上调度工作**

操作系统应该如何在多 CPU 上调度工作？会遇到什么新问题？已有的技术依旧适用吗？是否需要新的思路？

## 10.1　背景：多处理器架构

为了理解多处理器调度带来的新问题，必须先知道它与单 CPU 之间的基本区别。区别的核心在于对硬件缓存（cache）的使用（见图 10.1），以及多处理器之间共享数据的方式。本章将在较高层面讨论这些问题。更多信息可以其他地方找到[CSG99]，尤其是在高年级或

研究生计算机架构课程中。

在单 CPU 系统中，存在多级的硬件缓存（hardware cache），一般来说会让处理器更快地执行程序。缓存是很小但很快的存储设备，通常拥有内存中最热的数据的备份。相比之下，内存很大且拥有所有的数据，但访问速度较慢。通过将频繁访问的数据放在缓存中，系统似乎拥有又大又快的内存。

举个例子，假设一个程序需要从内存中加载指令并读取一个值，系统只有一个 CPU，拥有较小的缓存（如 64KB）和较大的内存。

程序第一次读取数据时，数据在内存中，因此需要花费较长的时间（可能数十或数百纳秒）。处理器判断该数据很可能会被再次使用，因此将其放入 CPU 缓存中。如果之后程序再次需要使用同样的数据，CPU 会先查找缓存。因为在缓存中找到了数据，所以取数据快得多（比如几纳秒），程序也就运行更快。

缓存是基于局部性（locality）的概念，局部性有两种，即时间局部性和空间局部性。时间局部性是指当一个数据被访问后，它很有可能会在不久的将来被再次访问，比如循环代码中的数据或指令本身。而空间局部性指的是，当程序访问地址为 $x$ 的数据时，很有可能会紧接着访问 $x$ 周围的数据，比如遍历数组或指令的顺序执行。由于这两种局部性存在于大多数的程序中，硬件系统可以很好地预测哪些数据可以放入缓存，从而运行得很好。

有趣的部分来了：如果系统有多个处理器，并共享同一个内存，如图 10.2 所示，会怎样呢？

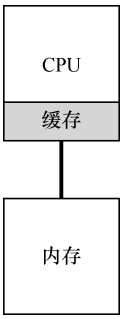



图 10.1 带缓存的单 CPU

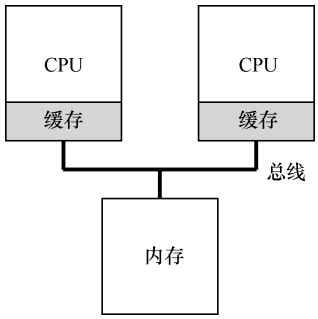



图 10.2 两个有缓存的 CPU 共享内存

事实证明，多 CPU 的情况下缓存要复杂得多。例如，假设一个运行在 CPU 1 上的程序从内存地址 A 读取数据。由于不在 CPU 1 的缓存中，所以系统直接访问内存，得到值 $D$。程序然后修改了地址 A 处的值，只是将它的缓存更新为新值 $D'$。将数据写回内存比较慢，因此系统（通常）会稍后再做。假设这时操作系统中断了该程序的运行，并将其交给 CPU 2，重新读取地址 A 的数据，由于 CPU 2 的缓存中并没有该数据，所以会直接从内存中读取，得到了旧值 $D$，而不是正确的值 $D'$。哎呀！

这一普遍的问题称为缓存一致性（cache coherence）问题，有大量的研究文献描述了解决这个问题时的微妙之处[SHW11]。这里我们会略过所有的细节，只提几个要点。选一门计算机体系结构课（或 3 门），你可以了解更多。

硬件提供了这个问题的基本解决方案：通过监控内存访问，硬件可以保证获得正确的数据，并保证共享内存的唯一性。在基于总线的系统中，一种方式是使用总线窥探（bus

snooping）[G83]。每个缓存都通过监听链接所有缓存和内存的总线，来发现内存访问。如果 CPU 发现对它放在缓存中的数据的更新，会作废（invalidate）本地副本（从缓存中移除），或更新（update）它（修改为新值）。回写缓存，如上面提到的，让事情更复杂（由于对内存的写入稍后才会看到），你可以想想基本方案如何工作。

## 10.2 别忘了同步

既然缓存已经做了这么多工作来提供一致性，应用程序（或操作系统）还需要关心共享数据的访问吗？依然需要！本书第 2 部分关于并发的描述中会详细介绍。虽然这里不会详细讨论，但我们会简单介绍（或复习）下其基本思路（假设你熟悉并发相关内容）。

跨 CPU 访问（尤其是写入）共享数据或数据结构时，需要使用互斥原语（比如锁），才能保证正确性（其他方法，如使用无锁（lock-free）数据结构，很复杂，偶尔才使用。详情参见并发部分关于死锁的章节）。例如，假设多 CPU 并发访问一个共享队列。如果没有锁，即使有底层一致性协议，并发地从队列增加或删除元素，依然不会得到预期结果。需要用锁来保证数据结构状态更新的原子性。

为了更具体，我们设想这样的代码序列，用于删除共享链表的一个元素，如图 10.3 所示。假设两个 CPU 上的不同线程同时进入这个函数。如果线程 1 执行第一行，会将 head 的当前值存入它的 tmp 变量。如果线程 2 接着也执行第一行，它也会将同样的 head 值存入它自己的私有 tmp 变量（tmp 在栈上分配，因此每个线程都有自己的私有存储）。因此，两个线程会尝试删除同一个链表头，而不是每个线程移除一个元素，这导致了各种问题（比如在第 4 行重复释放头元素，以及可能两次返回同一个数据）。

```
1    typedef struct __Node_t {
2        int value;
3        struct __Node_t *next;
4    } Node_t;
5
6    int List_Pop() {
7        Node_t *tmp = head;       // remember old head ...
8        int value = head->value;  // ... and its value
9        head = head->next;        // advance head to next pointer
10       free(tmp);                // free old head
11       return value;             // return value at head
12   }
```

图 10.3 简单的链表删除代码

当然，让这类函数正确工作的方法是加锁（locking）。这里只需要一个互斥锁（即 pthread_mutex_t m;），然后在函数开始时调用 lock(&m)，在结束时调用 unlock(&m)，确保代码的执行如预期。我们会看到，这里依然有问题，尤其是性能方面。具体来说，随着 CPU 数量的增加，访问同步共享的数据结构会变得很慢。

## 10.3 最后一个问题：缓存亲和度

在设计多处理器调度时遇到的最后一个问题，是所谓的缓存亲和度（cache affinity）。这个概念很简单：一个进程在某个 CPU 上运行时，会在该 CPU 的缓存中维护许多状态。下次该进程在相同 CPU 上运行时，由于缓存中的数据而执行得更快。相反，在不同的 CPU 上执行，会由于需要重新加载数据而很慢（好在硬件保证的缓存一致性可以保证正确执行）。因此多处理器调度应该考虑到这种缓存亲和性，并尽可能将进程保持在同一个 CPU 上。

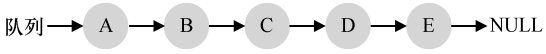
## 10.4 单队列调度

上面介绍了一些背景，现在来讨论如何设计一个多处理器系统的调度程序。最基本的方式是简单地复用单处理器调度的基本架构，将所有需要调度的工作放入一个单独的队列中，我们称之为单队列多处理器调度（Single Queue Multiprocessor Scheduling，SQMS）。这个方法最大的优点是简单。它不需要太多修改，就可以将原有的策略用于多个 CPU，选择最适合的工作来运行（例如，如果有两个 CPU，它可能选择两个最合适的工作）。

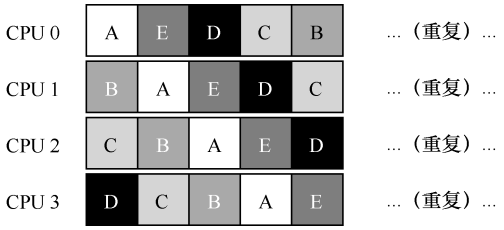
然而，SQMS 有几个明显的短板。第一个是缺乏可扩展性（scalability）。为了保证在多 CPU 上正常运行，调度程序的开发者需要在代码中通过加锁（locking）来保证原子性，如上所述。在 SQMS 访问单个队列时（如寻找下一个运行的工作），锁确保得到正确的结果。

然而，锁可能带来巨大的性能损失，尤其是随着系统中的 CPU 数增加时[A91]。随着这种单个锁的争用增加，系统花费了越来越多的时间在锁的开销上，较少的时间用于系统应该完成的工作（哪天在这里加上真正的测量数据就好了）。

SQMS 的第二个主要问题是缓存亲和性。比如，假设我们有 5 个工作（A、B、C、D、E）和 4 个处理器。调度队列如下：



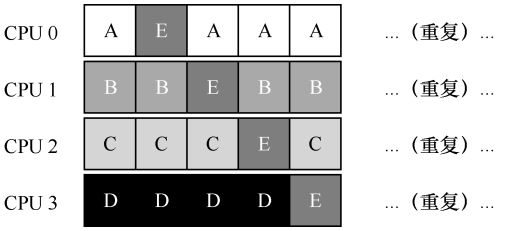

一段时间后，假设每个工作依次执行一个时间片，然后选择另一个工作，下面是每个 CPU 可能的调度序列：




由于每个 CPU 都简单地从全局共享的队列中选取下一个工作执行，因此每个工作都不断在不同 CPU 之间转移，这与缓存亲和的目标背道而驰。

为了解决这个问题，大多数 SQMS 调度程序都引入了一些亲和度机制，尽可能让进程

在同一个 CPU 上运行。保持一些工作的亲和度的同时，可能需要牺牲其他工作的亲和度来实现负载均衡。例如，针对同样的 5 个工作的调度如下：




这种调度中，A、B、C、D 这 4 个工作都保持在同一个 CPU 上，只有工作 E 不断地来回迁移（migrating），从而尽可能多地获得缓存亲和度。为了公平起见，之后我们可以选择不同的工作来迁移。但实现这种策略可能很复杂。

我们看到，SQMS 调度方式有优势也有不足。优势是能够从单 CPU 调度程序很简单地发展而来，根据定义，它只有一个队列。然而，它的扩展性不好（由于同步开销有限），并且不能很好地保证缓存亲和度。

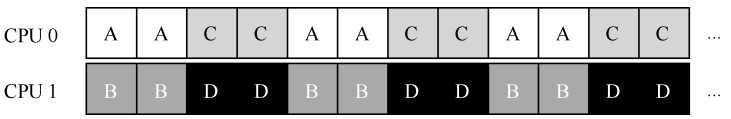
## 10.5 多队列调度

正是由于单队列调度程序的这些问题，有些系统使用了多队列的方案，比如每个 CPU 一个队列。我们称之为多队列多处理器调度（Multi-Queue Multiprocessor Scheduling，MQMS）

在 MQMS 中，基本调度框架包含多个调度队列，每个队列可以使用不同的调度规则，比如轮转或其他任何可能的算法。当一个工作进入系统后，系统会依照一些启发性规则（如随机或选择较空的队列）将其放入某个调度队列。这样一来，每个 CPU 调度之间相互独立，就避免了单队列的方式中由于数据共享及同步带来的问题。

例如，假设系统中有两个 CPU（CPU 0 和 CPU 1）。这时一些工作进入系统：A、B、C 和 D。由于每个 CPU 都有自己的调度队列，操作系统需要决定每个工作放入哪个队列。可能像下面这样做：


根据不同队列的调度策略，每个 CPU 从两个工作中选择，决定谁将运行。例如，利用轮转，调度结果可能如下所示：

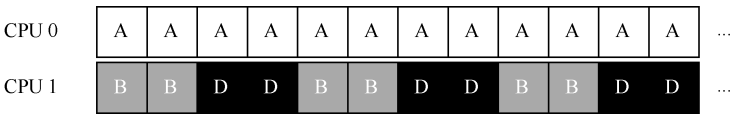



MQMS 比 SQMS 有明显的优势，它天生更具有可扩展性。队列的数量会随着 CPU 的增加而增加，因此锁和缓存争用的开销不是大问题。此外，MQMS 天生具有良好的缓存亲和度。所有工作都保持在固定的 CPU 上，因而可以很好地利用缓存数据。

但是，如果稍加注意，你可能会发现有一个新问题（这在多队列的方法中是根本的），即负载不均（load imbalance）。假定和上面设定一样（4 个工作，2 个 CPU），但假设一个工

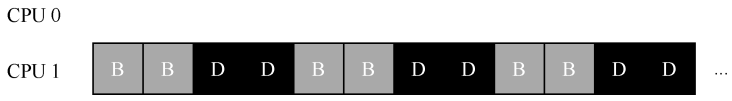
作（如 C）这时执行完毕。现在调度队列如下：


如果对系统中每个队列都执行轮转调度策略，会获得如下调度结果：




从图中可以看出，A 获得了 B 和 D 两倍的 CPU 时间，这不是期望的结果。更糟的是，假设 A 和 C 都执行完毕，系统中只有 B 和 D。调度队列看起来如下：


因此 CPU 使用时间线看起来令人难过：




所以可怜的多队列多处理器调度程序应该怎么办呢？怎样才能克服潜伏的负载不均问题，打败邪恶的……霸天虎军团[①]？如何才能不要问这些与这本好书几乎无关的问题？

**关键问题：如何应对负载不均**

多队列多处理器调度程序应该如何处理负载不均问题，从而更好地实现预期的调度目标？

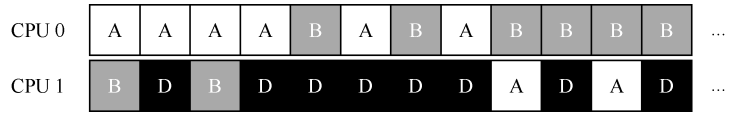
最明显的答案是让工作移动，这种技术我们称为迁移（migration）。通过工作的跨 CPU 迁移，可以真正实现负载均衡。

来看两个例子就更清楚了。同样，有一个 CPU 空闲，另一个 CPU 有一些工作。


在这种情况下，期望的迁移很容易理解：操作系统应该将 B 或 D 迁移到 CPU0。这次工作迁移导致负载均衡，皆大欢喜。

更棘手的情况是较早一些的例子，A 独自留在 CPU 0 上，B 和 D 在 CPU 1 上交替运行。


在这种情况下，单次迁移并不能解决问题。应该怎么做呢？答案是不断地迁移一个或多个工作。一种可能的解决方案是不断切换工作，如下面的时间线所示。可以看到，开始的时候 A 独享 CPU 0，B 和 D 在 CPU 1。一些时间片后，B 迁移到 CPU 0 与 A 竞争，D 则独享 CPU 1 一段时间。这样就实现了负载均衡。




① 一个鲜为人知的事实是，变形金刚的家乡塞伯坦星球被糟糕的 CPU 调度决策所摧毁。

当然，还有其他不同的迁移模式。但现在是最棘手的部分：系统如何决定发起这样的迁移？

一个基本的方法是采用一种技术，名为工作窃取（work stealing）[FLR98]。通过这种方法，工作量较少的（源）队列不定期地“偷看”其他（目标）队列是不是比自己的工作多。如果目标队列比源队列（显著地）更满，就从目标队列“窃取”一个或多个工作，实现负载均衡。

当然，这种方法也有让人抓狂的地方——如果太频繁地检查其他队列，就会带来较高的开销，可扩展性不好，而这是多队列调度最初的全部目标！相反，如果检查间隔太长，又可能会带来严重的负载不均。找到合适的阈值仍然是黑魔法，这在系统策略设计中很常见。

## 10.6 Linux 多处理器调度

有趣的是，在构建多处理器调度程序方面，Linux 社区一直没有达成共识。一直以来，存在 3 种不同的调度程序：O(1)调度程序、完全公平调度程序（CFS）以及 BF 调度程序（BFS）[①]。从 Meehean 的论文中可以找到对这些不同调度程序优缺点的对比总结[M11]。这里我们只总结一些基本知识。

O(1) CFS 采用多队列，而 BFS 采用单队列，这说明两种方法都可以成功。当然它们之间还有很多不同的细节。例如，O(1)调度程序是基于优先级的（类似于之前介绍的 MLFQ），随时间推移改变进程的优先级，然后调度最高优先级进程，来实现各种调度目标。交互性得到了特别关注。与之不同，CFS 是确定的比例调度方法（类似之前介绍的步长调度）。BFS 作为三个算法中唯一采用单队列的算法，也基于比例调度，但采用了更复杂的方案，称为最早最合适虚拟截止时间优先算法（EEVEF）[SA96]读者可以自己去了解这些现代操作系统的调度算法，现在应该能够理解它们的工作原理了！

## 10.7 小结

本章介绍了多处理器调度的不同方法。其中单队列的方式（SQMS）比较容易构建，负载均衡较好，但在扩展性和缓存亲和度方面有着固有的缺陷。多队列的方式（MQMS）有很好的扩展性和缓存亲和度，但实现负载均衡却很困难，也更复杂。无论采用哪种方式，都没有简单的答案：构建一个通用的调度程序仍是一项令人生畏的任务，因为即使很小的代码变动，也有可能导致巨大的行为差异。除非很清楚自己在做什么，或者有人付你很多钱，否则别干这种事。

## 参考资料

[A90]“The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors”Thomas E. Anderson IEEE TPDS Volume 1:1, January 1990

① 自己去查 BF 代表什么。预先警告，小心脏可能受不了。

这是一篇关于不同加锁方案扩展性好坏的经典论文。Tom Anderson 是非常著名的系统和网络研究者，也是一本非常好的操作系统教科书的作者。

[B+10]“An Analysis of Linux Scalability to Many Cores Abstract”
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich
OSDI ’10, Vancouver, Canada, October 2010
关于将 Linux 扩展到多核的很好的现代论文。

[CSG99]“Parallel Computer Architecture: A Hardware/Software Approach”David E. Culler, Jaswinder Pal Singh, and Anoop Gupta
Morgan Kaufmann, 1999
其中充满了并行机器和算法细节的宝藏。正如 Mark Hill 幽默地在书的护封上说的——这本书所包含的信息比大多数研究论文都多。

[FLR98]“The Implementation of the Cilk-5 Multithreaded Language”Matteo Frigo, Charles E. Leiserson, Keith Randall
PLDI ’98, Montreal, Canada, June 1998
Cilk 是用于编写并行程序的轻量级语言和运行库，并且是工作窃取范式的极好例子。

[G83]“Using Cache Memory To Reduce Processor-Memory Traffic”James R. Goodman
ISCA ’83, Stockholm, Sweden, June 1983
关于如何使用总线监听，即关注总线上看到的请求，构建高速缓存一致性协议的开创性论文。Goodman 在威斯康星的多年研究工作充满了智慧，这只是一个例子。

[M11]“Towards Transparent CPU Scheduling”Joseph T. Meehean
Doctoral Dissertation at University of Wisconsin—Madison, 2011
一篇涵盖了现代 Linux 多处理器调度如何工作的许多细节的论文。非常棒！但是，作为 Joe 的联合导师，我们可能在这里有点偏心。

[SHW11]“A Primer on Memory Consistency and Cache Coherence”Daniel J. Sorin, Mark D. Hill, and David A. Wood
Synthesis Lectures in Computer Architecture
Morgan and Claypool Publishers, May 2011
内存一致性和多处理器缓存的权威概述。对于喜欢对该主题深入了解的人来说，这是必读物。

[SA96]“Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Pro- portional Share Resource Allocation”
Ion Stoica and Hussein Abdel-Wahab
Technical Report TR-95-22, Old Dominion University, 1996
来自 Ion Stoica 的一份技术报告，其中介绍了很酷的调度思想。他现在是 U.C.伯克利大学的教授，也是网络、分布式系统和其他许多方面的世界级专家。