第 9 章　调度：比例份额

在本章中，我们来看一个不同类型的调度程序——比例份额（proportional-share）调度程序，有时也称为公平份额（fair-share）调度程序。比例份额算法基于一个简单的想法：调度程序的最终目标，是确保每个工作获得一定比例的 CPU 时间，而不是优化周转时间和响应时间。

比例份额调度程序有一个非常优秀的现代例子，由 Waldspurger 和 Weihl 发现，名为彩票调度（lottery scheduling） [WW94]。但这个想法其实出现得更早[KL88]。基本思想很简单：每隔一段时间，都会举行一次彩票抽奖，以确定接下来应该运行哪个进程。越是应该频繁运行的进程，越是应该拥有更多地赢得彩票的机会。很简单吧？现在，谈谈细节！但还是先看看下面的关键问题。

关键问题：如何按比例分配 CPU

如何设计调度程序来按比例分配 CPU？其关键的机制是什么？效率如何？

9.1　基本概念：彩票数表示份额

彩票调度背后是一个非常基本的概念：彩票数（ticket）代表了进程（或用户或其他）占有某个资源的份额。一个进程拥有的彩票数占总彩票数的百分比，就是它占有资源的份额。

下面来看一个例子。假设有两个进程 A 和 B，A 拥有 75 张彩票，B 拥有 25 张。因此我们希望 A 占用 75%的 CPU 时间，而 B 占用 25%。

通过不断定时地（比如，每个时间片）抽取彩票，彩票调度从概率上（但不是确定的）获得这种份额比例。抽取彩票的过程很简单：调度程序知道总共的彩票数（在我们的例子中，有 100 张）。调度程序抽取中奖彩票，这是从 0 和 99[①]之间的一个数，拥有这个数对应的彩票的进程中奖。假设进程 A 拥有 0 到 74 共 75 张彩票，进程 B 拥有 75 到 99 的 25 张，中奖的彩票就决定了运行 A 或 B。调度程序然后加载中奖进程的状态，并运行它。

提示：利用随机性

彩票调度最精彩的地方在于利用了随机性（randomness）。当你需要做出决定时，采用随机的方式常常是既可靠又简单的选择。

随机方法相对于传统的决策方式，至少有 3 点优势。第一，随机方法常常可以避免奇怪的边角情况，

① 计算机科学家总是从 0 开始计数。对于非计算机类型的人来说，这非常奇怪，所以著名人士不得不撰文说明这样做的原因[D82]。

较传统的算法可能在处理这些情况时遇到麻烦。例如 LRU 替换策略（稍后会在虚拟内存的章节详细介绍）。虽然 LRU 通常是很好的替换算法，但在有重复序列的负载时表现非常差。但随机方法就没有这种最差情况。

第二，随机方法很轻量，几乎不需要记录任何状态。在传统的公平份额调度算法中，记录每个进程已经获得了多少的 CPU 时间，需要对每个进程计时，这必须在每次运行结束后更新。而采用随机方式后每个进程只需要非常少的状态（即每个进程拥有的彩票号码）。

第三，随机方法很快。只要能很快地产生随机数，做出决策就很快。因此，随机方式在对运行速度要求高的场景非常适用。当然，越是需要快的计算速度，随机就会越倾向于伪随机。

下面是彩票调度程序输出的中奖彩票：

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 49
```

下面是对应的调度结果:

```
A    A  A      A  A  A  A  A  A      A      A  A  A  A  A  A
   B         B                   B     B
```

从这个例子中可以看出，彩票调度中利用了随机性，这导致了从概率上满足期望的比例，但并不能确保。在上面的例子中，工作 B 运行了 20 个时间片中的 4 个，只是占了 20%，而不是期望的 25%。但是，这两个工作运行得时间越长，它们得到的 CPU 时间比例就会越接近期望。

提示：用彩票来表示份额

彩票（步长）调度的设计中，最强大（且最基本）的机制是彩票。在这些例子中，彩票用于表示一个进程占有 CPU 的份额，但也可以用在更多的地方。比如在虚拟机管理程序的虚存管理的最新研究工作中，Waldspurger 提出了用彩票来表示用户占用操作系统内存份额的方法[W02]。因此，如果你需要通过什么机制来表示所有权比例，这个概念可能就是彩票。

9.2 彩票机制

彩票调度还提供了一些机制，以不同且有效的方式来调度彩票。一种方式是利用彩票货币（ticket currency）的概念。这种方式允许拥有一组彩票的用户以他们喜欢的某种货币，将彩票分给自己的不同工作。之后操作系统再自动将这种货币兑换为正确的全局彩票。

比如，假设用户 A 和用户 B 每人拥有 100 张彩票。用户 A 有两个工作 A1 和 A2，他以自己的货币，给每个工作 500 张彩票（共 1000 张）。用户 B 只运行一个工作，给它 10 张彩票（总共 10 张）。操作系统将进行兑换，将 A1 和 A2 拥有的 A 的货币 500 张，兑换成全局货币 50 张。类似地，兑换给 B1 的 10 张彩票兑换成 100 张。然后会对全局彩票货币（共 200 张）举行抽奖，决定哪个工作运行。

```
User A -> 500 (A's currency) to A1 -> 50 (global currency)
       -> 500 (A's currency) to A2 -> 50 (global currency)
User B -> 10 (B's currency) to B1 -> 100 (global currency)
```

另一个有用的机制是彩票转让（ticket transfer）。通过转让，一个进程可以临时将自己的彩票交给另一个进程。这种机制在客户端/服务端交互的场景中尤其有用，在这种场景中，客户端进程向服务端发送消息，请求其按自己的需求执行工作，为了加速服务端的执行，客户端可以将自己的彩票转让给服务端，从而尽可能加速服务端执行自己请求的速度。服务端执行结束后会将这部分彩票归还给客户端。

最后，彩票通胀（ticket inflation）有时也很有用。利用通胀，一个进程可以临时提升或降低自己拥有的彩票数量。当然在竞争环境中，进程之间互相不信任，这种机制就没什么意义。一个贪婪的进程可能给自己非常多的彩票，从而接管机器。但是，通胀可以用于进程之间相互信任的环境。在这种情况下，如果一个进程知道它需要更多 CPU 时间，就可以增加自己的彩票，从而将自己的需求告知操作系统，这一切不需要与任何其他进程通信。

9.3 实现

彩票调度中最不可思议的，或许就是实现简单。只需要一个不错的随机数生成器来选择中奖彩票和一个记录系统中所有进程的数据结构（一个列表），以及所有彩票的总数。

假定我们用列表记录进程。下面的例子中有 A、B、C 这 3 个进程，每个进程有一定数量的彩票。

在做出调度决策之前，首先要从彩票总数 400 中选择一个随机数（中奖号码）[①]。假设选择了 300。然后，遍历链表，用一个简单的计数器帮助我们找到中奖者（见图 9.1）。

```
1    // counter: used to track if we've found the winner yet
2    int counter = 0;
3
4    // winner: use some call to a random number generator to
5    //         get a value, between 0 and the total # of tickets
6    int winner = getrandom(0, totaltickets);
7
8    // current: use this to walk through the list of jobs
9    node_t *current = head;
10
11   // loop until the sum of ticket values is > the winner
12   while (current) {
13       counter = counter + current->tickets;
14       if (counter > winner)
15           break; // found the winner
16       current = current->next;
17   }
18   // 'current' is the winner: schedule it...
```

图 9.1 彩票调度决定代码

① 令人惊讶的是，正如 Björn Lindberg 所指出的那样，要做对，这可能是一个挑战。

这段代码从前向后遍历进程列表，将每张票的值加到 counter 上，直到值超过 winner。这时，当前的列表元素所对应的进程就是中奖者。在我们的例子中，中奖彩票是 300。首先，计 A 的票后，counter 增加到 100。因为 100 小于 300，继续遍历。然后 counter 会增加到 150（B 的彩票），仍然小于 300，继续遍历。最后，counter 增加到 400（显然大于 300），因此退出遍历，current 指向 C（中奖者）。

要让这个过程更有效率，建议将列表项按照彩票数递减排序。这个顺序并不会影响算法的正确性，但能保证用最小的迭代次数找到需要的节点，尤其当大多数彩票被少数进程掌握时。

9.4 一个例子

为了更好地理解彩票调度的运行过程，我们现在简单研究一下两个互相竞争工作的完成时间，每个工作都有相同数目的 100 张彩票，以及相同的运行时间 *R*（稍后会改变）。

这种情况下，我们希望两个工作在大约同时完成，但由于彩票调度算法的随机性，有时一个工作会先于另一个完成。为了量化这种区别，我们定义了一个简单的不公平指标 *U*（unfairness metric），将两个工作完成时刻相除得到 *U* 的值。比如，运行时间 *R* 为 10，第一个工作在时刻 10 完成，另一个在 20，*U*=10/20=0.5。如果两个工作几乎同时完成，*U* 的值将很接近于 1。在这种情况下，我们的目标是：完美的公平调度程序可以做到 *U*=1。

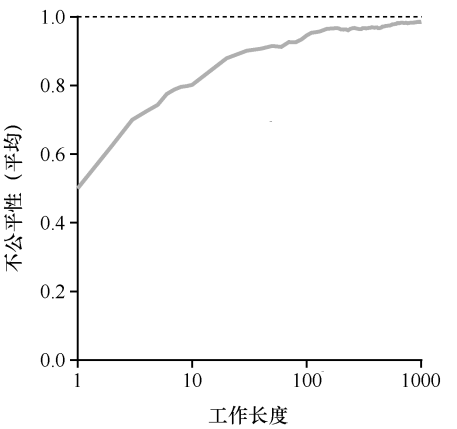

图 9.2 彩票公平性研究

图 9.2 展示了当两个工作的运行时间从 1 到 1000 变化时，30 次试验的平均 *U* 值（利用本章末尾的模拟器产生的结果）。可以看出，当工作执行时间很短时，平均不公平度非常糟糕。只有当工作执行非常多的时间片时，彩票调度算法才能得到期望的结果。

9.5 如何分配彩票

关于彩票调度，还有一个问题没有提到，那就是如何为工作分配彩票？这是一个非常棘手的问题，系统的运行严重依赖于彩票的分配。假设用户自己知道如何分配，因此可以给每个用户一定量的彩票，由用户按照需要自主分配给自己的工作。然而这种方案似乎什么也没有解决——还是没有给出具体的分配策略。因此对于给定的一组工作，彩票分配的问题依然没有最佳答案。

9.6 为什么不是确定的

你可能还想知道，究竟为什么要利用随机性？从上面的内容可以看出，虽然随机方式可以使得调度程序的实现简单（且大致正确），但偶尔并不能产生正确的比例，尤其在工作运行时间很短的情况下。由于这个原因，Waldspurger 提出了步长调度（stride scheduling），一个确定性的公平分配算法[W95]。

步长调度也很简单。系统中的每个工作都有自己的步长，这个值与票数值成反比。在上面的例子中，A、B、C 这 3 个工作的票数分别是 100、50 和 250，我们通过用一个大数分别除以他们的票数来获得每个进程的步长。比如用 10000 除以这些票数值，得到了 3 个进程的步长分别为 100、200 和 40。我们称这个值为每个进程的步长（stride）。每次进程运行后，我们会让它的计数器 [称为行程（pass）值] 增加它的步长，记录它的总体进展。

之后，调度程序使用进程的步长及行程值来确定调度哪个进程。基本思路很简单：当需要进行调度时，选择目前拥有最小行程值的进程，并且在运行之后将该进程的行程值增加一个步长。下面是 Waldspurger[W95]给出的伪代码：

```
current = remove_min(queue);       // pick client with minimum pass
schedule(current);                 // use resource for quantum
current->pass += current->stride;  // compute next pass using stride
insert(queue, current);            // put back into the queue
```

在我们的例子中，3 个进程（A、B、C）的步长值分别为 100、200 和 40，初始行程值都为 0。因此，最初，所有进程都可能被选择执行。假设选择 A（任意的，所有具有同样低的行程值的进程，都可能被选中）。A 执行一个时间片后，更新它的行程值为 100。然后运行 B，并更新其行程值为 200。最后执行 C，C 的行程值变为 40。这时，算法选择最小的行程值，是 C，执行并增加为 80（C 的步长是 40）。然后 C 再次运行（依然行程值最小），行程值增加到 120。现在运行 A，更新它的行程值为 200（现在与 B 相同）。然后 C 再次连续运行两次，行程值也变为 200。此时，所有行程值再次相等，这个过程会无限地重复下去。表 9.1 展示了一段时间内调度程序的行为。

表 9.1 步长调度：记录

行程值（A）（步长=100）	行程值（B）（步长=200）	行程值（C）（步长=40）	谁运行
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	……

可以看出，C 运行了 5 次、A 运行了 2 次，B 一次，正好是票数的比例——200、100 和 50。彩票调度算法只能一段时间后，在概率上实现比例，而步长调度算法可以在每个调度周期后做到完全正确。

你可能想知道，既然有了可以精确控制的步长调度算法，为什么还要彩票调度算法呢？好吧，彩票调度有一个步长调度没有的优势——不需要全局状态。假如一个新的进程在上面的步长调度执行过程中加入系统，应该怎么设置它的行程值呢？设置成 0 吗？这样的话，它就独占 CPU 了。而彩票调度算法不需要对每个进程记录全局状态，只需要用新进程的票数更新全局的总票数就可以了。因此彩票调度算法能够更合理地处理新加入的进程。

9.7 小结

本章介绍了比例份额调度的概念，并简单讨论了两种实现：彩票调度和步长调度。彩票调度通过随机值，聪明地做到了按比例分配。步长调度算法能够确定的获得需要的比例。虽然两者都很有趣，但由于一些原因，并没有作为 CPU 调度程序被广泛使用。一个原因是这两种方式都不能很好地适合 I/O[AC97]；另一个原因是其中最难的票数分配问题并没有确定的解决方式，例如，如何知道浏览器进程应该拥有多少票数？通用调度程序（像前面讨论的 MLFQ 及其他类似的 Linux 调度程序）做得更好，因此得到了广泛的应用。

结果，比例份额调度程序只有在这些问题可以相对容易解决的领域更有用（例如容易确定份额比例）。例如在虚拟（virtualized）数据中心中，你可能会希望分配 1/4 的 CPU 周期给 Windows 虚拟机，剩余的给 Linux 系统，比例分配的方式可以更简单高效。详细信息请参考 Waldspurger [W02]，该文介绍了 VMWare 的 ESX 系统如何用比例分配的方式来共享内存。

参考资料

[AC97]“Extending Proportional-Share Scheduling to a Network of Workstations”Andrea C. Arpaci-Dusseau and David E. Culler

PDPTA’97, June 1997

这是本书的一位作者撰写的论文，关于如何扩展比例共享调度，从而在群集环境中更好地工作。

[D82]“Why Numbering Should Start At Zero”
Edsger Dijkstra, August 1982
来自计算机科学先驱之一 E. Dijkstra 的简短讲义。在关于并发的部分，我们会听到更多关于 E. Dijkstra 的信息。与此同时，请阅读这份讲义，其中有一句激励人心的话：“我的一个同事（不是一个计算科学家）指责一些年轻的计算科学家‘卖弄学问’，因为他们从零开始编号。”该讲义解释了为什么这样做是合理的。

[KL88]“A Fair Share Scheduler”
J．Kay and P. Lauder
CACM, Volume 31 Issue 1, January 1988
关于公平份额调度程序的早期参考文献。

[WW94]“Lottery Scheduling: Flexible Proportional-Share Resource Management”Carl A. Waldspurger and William E. Weihl
OSDI ’94, November 1994
关于彩票调度的里程碑式的论文，让调度、公平分享和简单随机算法的力量在操作系统社区重新焕发了活力。

[W95]“Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management”Carl A. Waldspurger
Ph.D. Thesis, MIT, 1995
Waldspurger 的获奖论文，概述了彩票和步长调度。如果你想写一篇博士论文，总应该有一个很好的例子，让你有个努力的方向：这是一个很好的例子。

[W02]“Memory Resource Management in VMware ESX Server”Carl A. Waldspurger
OSDI ’02, Boston, Massachusetts
关于 VMM（虚拟机管理程序）中的内存管理的文章。除了相对容易阅读之外，该论文还包含许多有关新型 VMM 层面内存管理的很酷的想法。

作业

lottery.py 这个程序允许你查看彩票调度程序的工作原理。详情请参阅 README 文件。

问题

1．计算 3 个工作在随机种子为 1、2 和 3 时的模拟解。

2．现在运行两个具体的工作：每个长度为 10，但是一个（工作 0）只有一张彩票，另一个（工作 1）有 100 张（-l 10：1,10：100）。

彩票数量如此不平衡时会发生什么？在工作 1 完成之前，工作 0 是否会运行？多久？一般来说，这种彩票不平衡对彩票调度的行为有什么影响？

3．如果运行两个长度为 100 的工作，都有 100 张彩票（–l100：100,100：100），调度程序有多不公平？运行一些不同的随机种子来确定（概率上的）答案。不公平性取决于一项工作比另一项工作早完成多少。

4．随着量子规模（-q）变大，你对上一个问题的答案如何改变？

5．你可以制作类似本章中的图表吗？

还有什么值得探讨的？用步长调度程序，图表看起来如何？