的优先情绪而已，而是根据观察到的行为调整它的优先级。例如，如果一个工作不断放弃 CPU 去等待键盘输入，这是交互型进程的可能行为，MLFQ 因此会让它保持高优先级。相反，如果一个工作长时间地占用 CPU，MLFQ 会降低其优先级。通过这种方式，MLFQ 在进程运行过程中学习其行为，从而利用工作的历史来预测它未来的行为。

至此，我们得到了 MLFQ 的两条基本规则。

- **规则 1**：如果 A 的优先级 ＞B 的优先级，运行 A（不运行 B）。
- **规则 2**：如果 A 的优先级 ＝B 的优先级，轮转运行 A 和 B。

如果要在某个特定时刻展示队列，可能会看到如下内容（见图 8.1）。图 8.1 中，最高优先级有两个工作（A 和 B），工作 C 位于中等优先级，而 D 的优先级最低。按刚才介绍的基本规则，由于 A 和 B 有最高优先级，调度程序将交替的调度他们，可怜的 C 和 D 永远都没有机会运行，太气人了！

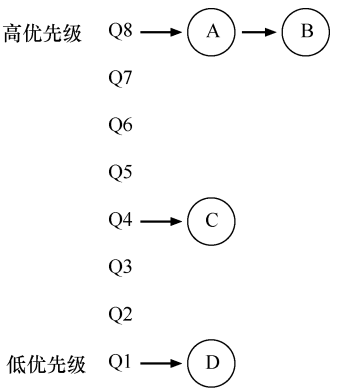

图 8.1 MLFQ 的例子

当然，这只是展示了一些队列的静态快照，并不能让你真正明白 MLFQ 的工作原理。我们需要理解工作的优先级如何随时间变化。初次拿起本书阅读一章的人可能会吃惊，这正是我们接下来要做的事。

8.2 尝试 1：如何改变优先级

我们必须决定，在一个工作的生命周期中，MLFQ 如何改变其优先级（在哪个队列中）。要做到这一点，我们必须记得工作负载：既有运行时间很短、频繁放弃 CPU 的交互型工作，也有需要很多 CPU 时间、响应时间却不重要的长时间计算密集型工作。下面是我们第一次尝试优先级调整算法。

- **规则 3**：工作进入系统时，放在最高优先级（最上层队列）。
- **规则 4a**：工作用完整个时间片后，降低其优先级（移入下一个队列）。
- **规则 4b**：如果工作在其时间片以内主动释放 CPU，则优先级不变。

实例 1：单个长工作

我们来看一些例子。首先，如果系统中有一个需要长时间运行的工作，看看会发生什么。图 8.2 展示了在一个有 3 个队列的调度程序中，随着时间的推移，这个工作的运行情况。

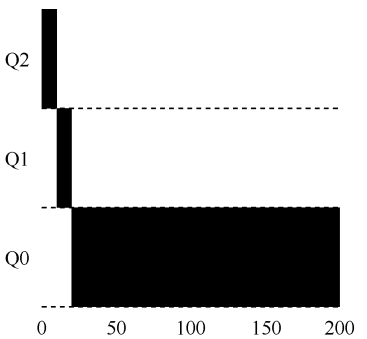

图 8.2 长时间工作随时间的变化

从这个例子可以看出，该工作首先进入最高优先级（Q2）。执行一个 10ms 的时间片后，调度程序将工作的优先级减 1，因此进入 Q1。在 Q1 执行一个时间片后，最终降低优先级进入系统的最低优先级（Q0），一直留在那里。相当简单，不是吗？

实例 2：来了一个短工作

再看一个较复杂的例子，看看 MLFQ 如何近似 SJF。在这个例子中，有两个工作：A 是一个长时间运行的 CPU 密集型工作，B 是一个运行时间很短的交互型工作。假设 A 执行一段时间后 B 到达。会发生什么呢？对 B 来说，MLFQ 会近似于 SJF 吗？

图 8.3 展示了这种场景的结果。A（用黑色表示）在最低优先级队列执行（长时间运行的 CPU 密集型工作都这样）。B（用灰色表示）在时间 T=100 时到达，并被加入最高优先级队列。由于它的运行时间很短（只有 20ms），经过两个时间片，在被移入最低优先级队列之前，B 执行完毕。然后 A 继续运行（在低优先级）。

通过这个例子，你大概可以体会到这个算法的一个主要目标：如果不知道工作是短工作还是长工作，那么就在开始的时候假设其是短工作，并赋予最高优先级。如果确实是短工作，则很快会执行完毕，否则将被慢慢移入低优先级队列，而这时该工作也被认为是长工作了。通过这种方式，MLFQ 近似于 SJF。

实例 3：如果有 I/O 呢

看一个有 I/O 的例子。根据上述规则 4b，如果进程在时间片用完之前主动放弃 CPU，则保持它的优先级不变。这条规则的意图很简单：假设交互型工作中有大量的 I/O 操作（比如等待用户的键盘或鼠标输入），它会在时间片用完之前放弃 CPU。在这种情况下，我们不想处罚它，只是保持它的优先级不变。

图 8.4 展示了这个运行过程，交互型工作 B（用灰色表示）每执行 1ms 便需要进行 I/O 操作，它与长时间运行的工作 A（用黑色表示）竞争 CPU。MLFQ 算法保持 B 在最高优先级，因为 B 总是让出 CPU。如果 B 是交互型工作，MLFQ 就进一步实现了它的目标，让交互型工作快速运行。

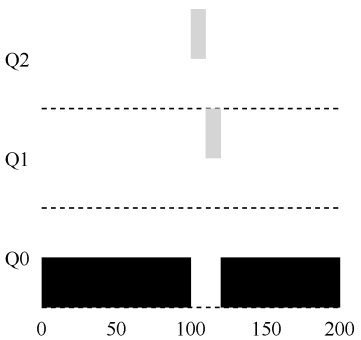

图 8.3 一个交互型工作

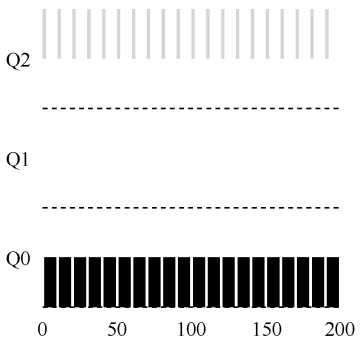

图 8.4 混合 I/O 密集型和 CPU 密集型工作负载

当前 MLFQ 的一些问题

至此，我们有了基本的 MLFQ。它看起来似乎相当不错，长工作之间可以公平地分享 CPU，又能给短工作或交互型工作很好的响应时间。然而，这种算法有一些非常严重的缺点。

你能想到吗？

（暂停一下，尽量让脑筋转转弯）

首先，会有饥饿（starvation）问题。如果系统有“太多”交互型工作，就会不断占用 CPU，导致长工作永远无法得到 CPU（它们饿死了）。即使在这种情况下，我们希望这些长工作也能有所进展。

其次，聪明的用户会重写程序，愚弄调度程序（game the scheduler）。愚弄调度程序指的是用一些卑鄙的手段欺骗调度程序，让它给你远超公平的资源。上述算法对如下的攻击束手无策：进程在时间片用完之前，调用一个 I/O 操作（比如访问一个无关的文件），从而主动释放 CPU。如此便可以保持在高优先级，占用更多的 CPU 时间。做得好时（比如，每运行 99%的时间片时间就主动放弃一次 CPU），工作可以几乎独占 CPU。

最后，一个程序可能在不同时间表现不同。一个计算密集的进程可能在某段时间表现为一个交互型的进程。用我们目前的方法，它不会享受系统中其他交互型工作的待遇。

8.3 尝试 2：提升优先级

让我们试着改变之前的规则，看能否避免饥饿问题。要让 CPU 密集型工作也能取得一些进展（即使不多），我们能做些什么？

一个简单的思路是周期性地提升（boost）所有工作的优先级。可以有很多方法做到，但我们就用最简单的：将所有工作扔到最高优先级队列。于是有了如下的新规则。

- **规则 5**：经过一段时间 *S*，就将系统中所有工作重新加入最高优先级队列。

新规则一下解决了两个问题。首先，进程不会饿死——在最高优先级队列中，它会以轮转的方式，与其他高优先级工作分享 CPU，从而最终获得执行。其次，如果一个 CPU 密集型工作变成了交互型，当它优先级提升时，调度程序会正确对待它。

我们来看一个例子。在这种场景下，我们展示长工作与两个交互型短工作竞争 CPU 时的行为。图 8.5 包含两张图。左边没有优先级提升，长工作在两个短工作到达后被饿死。右边每 50ms 就有一次优先级提升（这里只是举例，这个值可能过小），因此至少保证长工作会有一些进展，每过 50ms 就被提升到最高优先级，从而定期获得执行。

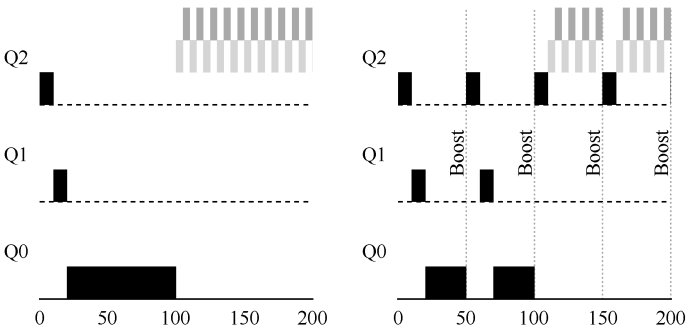

图 8.5 不采用优先级提升（左）和采用（右）

当然，添加时间段 *S* 导致了明显的问题：*S* 的值应该如何设置？德高望重的系统研究员

John Ousterhout[O11]曾将这种值称为“巫毒常量（voo-doo constant）”，因为似乎需要一些黑魔法才能正确设置。如果 *S* 设置得太高，长工作会饥饿；如果设置得太低，交互型工作又得不到合适的 CPU 时间比例。

8.4 尝试 3：更好的计时方式

现在还有一个问题要解决：如何阻止调度程序被愚弄？可以看出，这里的元凶是规则 4a 和 4b，导致工作在时间片以内释放 CPU，就保留它的优先级。那么应该怎么做？

这里的解决方案，是为 MLFQ 的每层队列提供更完善的 CPU 计时方式（accounting）。调度程序应该记录一个进程在某一层中消耗的总时间，而不是在调度时重新计时。只要进程用完了自己的配额，就将它降到低一优先级的队列中去。不论它是一次用完的，还是拆成很多次用完。因此，我们重写规则 4a 和 4b。

- **规则 4**：一旦工作用完了其在某一层中的时间配额（无论中间主动放弃了多少次 CPU），就降低其优先级（移入低一级队列）。

来看一个例子。图 8.6 对比了在规则 4a、4b 的策略下（左图），以及在新的规则 4（右图）的策略下，同样试图愚弄调度程序的进程的表现。没有规则 4 的保护时，进程可以在每个时间片结束前发起一次 I/O 操作，从而垄断 CPU 时间。有了这样的保护后，不论进程的 I/O 行为如何，都会慢慢地降低优先级，因而无法获得超过公平的 CPU 时间比例。

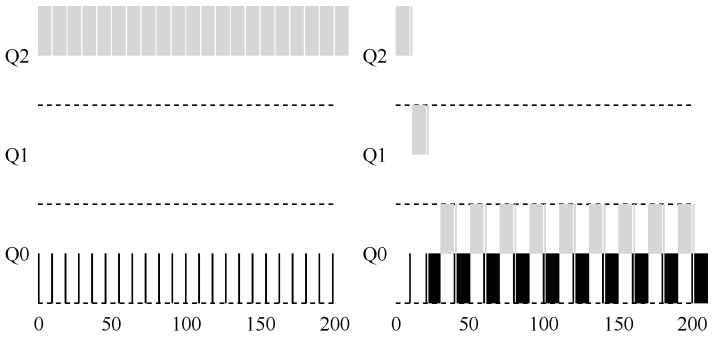

图 8.6 不采用愚弄反制（左）和采用（右）

8.5 MLFQ 调优及其他问题

关于 MLFQ 调度算法还有一些问题。其中一个大问题是如何配置一个调度程序，例如，配置多少队列？每一层队列的时间片配置多大？为了避免饥饿问题以及进程行为改变，应该多久提升一次进程的优先级？这些问题都没有显而易见的答案，因此只有利用对工作负载的经验，以及后续对调度程序的调优，才会导致令人满意的平衡。

例如，大多数的 MLFQ 变体都支持不同队列可变的时间片长度。高优先级队列通常只有较短的时间片（比如 10ms 或者更少），因而这一层的交互工作可以更快地切换。相反，

低优先级队列中更多的是 CPU 密集型工作，配置更长的时间片会取得更好的效果。图 8.7 展示了一个例子，两个长工作在高优先级队列执行 10ms，中间队列执行 20ms，最后在最低优先级队列执行 40ms。

提示：避免巫毒常量（Ousterhout 定律）

尽可能避免巫毒常量是个好主意。然而，从上面的例子可以看出，这通常很难。当然，我们也可以让系统自己去学习一个很优化的值，但这同样也不容易。因此，通常我们会有一个写满各种参数值默认值的配置文件，使得系统管理员可以方便地进行修改调整。然而，大多数使用者并不会去修改这些默认值，这时就寄希望于默认值合适了。这个提示是由资深的 OS 教授 John Ousterhout 提出的，因此称为 Ousterhout 定律（Ousterhout's Law）。

Solaris 的 MLFQ 实现（时分调度类 TS）很容易配置。它提供了一组表来决定进程在其生命周期中如何调整优先级，每层的时间片多大，以及多久提升一个工作的优先级[AD00]。管理员可以通过这些表，让调度程序的行为方式不同。该表默认有 60 层队列，时间片长度从 20ms（最高优先级），到几百 ms（最低优先级），每一秒左右提升一次进程的优先级。

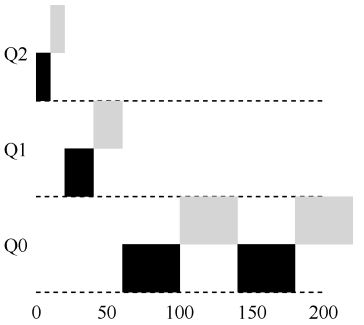

图 8.7 优先级越低，时间片越长

其他一些 MLFQ 调度程序没用表，甚至没用本章中讲到的规则，有些采用数学公式来调整优先级。例如，FreeBSD 调度程序（4.3 版本），会基于当前进程使用了多少 CPU，通过公式计算某个工作的当前优先级[LM+89]。另外，使用量会随时间衰减，这提供了期望的优先级提升，但与这里描述方式不同。阅读 Epema 的论文，他漂亮地概括了这种使用量衰减（decay-usage）算法及其特征[E95]。

最后，许多调度程序有一些我们没有提到的特征。例如，有些调度程序将最高优先级队列留给操作系统使用，因此通常的用户工作是无法得到系统的最高优先级的。有些系统允许用户给出优先级设置的建议（advice），比如通过命令行工具 nice，可以增加或降低工作的优先级（稍微），从而增加或降低它在某个时刻运行的机会。更多信息请查看 man 手册。

8.6 MLFQ：小结

本章介绍了一种调度方式，名为多级反馈队列（MLFQ）。你应该已经知道它为什么叫这个名字——它有多级队列，并利用反馈信息决定某个工作的优先级。以史为鉴：关注进程的一贯表现，然后区别对待。

提示：尽可能多地使用建议

操作系统很少知道什么策略对系统中的单个进程和每个进程算是好的，因此提供接口并允许用户或管理员给操作系统一些提示（hint）常常很有用。我们通常称之为建议（advice），因为操作系统不一定要关注它，但是可能会将建议考虑在内，以便做出更好的决定。这种用户建议的方式在操作系统中的各个领域经常十分有用，包括调度程序（通过 nice）、内存管理（madvise），以及文件系统（通知预取和缓存[P+95]）。

本章包含了一组优化的 MLFQ 规则。为了方便查阅，我们重新列在这里。

- **规则 1**：如果 A 的优先级 ＞B 的优先级，运行 A（不运行 B）。
- **规则 2**：如果 A 的优先级 ＝B 的优先级，轮转运行 A 和 B。
- **规则 3**：工作进入系统时，放在最高优先级（最上层队列）。
- **规则 4**：一旦工作用完了其在某一层中的时间配额（无论中间主动放弃了多少次 CPU），就降低其优先级（移入低一级队列）。
- **规则 5**：经过一段时间 *S*，就将系统中所有工作重新加入最高优先级队列。

MLFQ 有趣的原因是：它不需要对工作的运行方式有先验知识，而是通过观察工作的运行来给出对应的优先级。通过这种方式，MLFQ 可以同时满足各种工作的需求：对于短时间运行的交互型工作，获得类似于 SJF/STCF 的很好的全局性能，同时对长时间运行的 CPU 密集型负载也可以公平地、不断地稳步向前。因此，许多系统使用某种类型的 MLFQ 作为自己的基础调度程序，包括类 BSD UNIX 系统[LM+89，B86]、Solaris[M06]以及 Windows NT 和其后的 Window 系列操作系统。

参考资料

[AD00]“Multilevel Feedback Queue Scheduling in Solaris”Andrea Arpaci-Dusseau
本书的一位作者就 Solaris 调度程序的细节做了一些简短的说明。我们这里的描述可能有失偏颇，但这些讲义还是不错的。

[B86]“The Design of the UNIX Operating System”
M.J. Bach
Prentice-Hall, 1986
关于如何构建真正的 UNIX 操作系统的经典老书之一。对内核黑客来说，这是必读内容。

[C+62]“An Experimental Time-Sharing System”
F. J. Corbato, M. M. Daggett, R. C. Daley IFIPS 1962
有点难读，但这是多级反馈调度中许多首创想法的来源。其中大部分后来进入了 Multics，人们可以争辩说它是有史以来有影响力的操作系统。

[CS97]“Inside Windows NT”
Helen Custer and David A. Solomon Microsoft Press, 1997
如果你想了解 UNIX 以外的东西，来读 NT 书吧！当然，你为什么会想？好吧，我们在开玩笑吧。说不定有一天你会为微软工作。

[E95]“An Analysis of Decay-Usage Scheduling in Multiprocessors”
D.H.J.　Epema SIGMETRICS ’95
一篇关于 20 世纪 90 年代中期调度技术发展状况的优秀论文，概述了使用量衰减调度程序背后的基本方法。

[LM+89]“The Design and Implementation of the 4.3BSD UNIX Operating System”

S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman Addison-Wesley, 1989
另一本操作系统经典图书，由 BSD 背后的 4 个主要人员编写。本书后面的版本虽然更新了，但感觉不如这一版好。

[M06]“Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture”Richard McDougall
Prentice-Hall, 2006
一本关于 Solaris 及其工作原理的好书。

[O11]“John Ousterhout’s Home Page”John Ousterhout
著名的 Ousterhout 教授的主页。本书的两位合著者一起在研究生院学习 Ousterhout 的研究生操作系统课程。事实上，这是两位合著者相互认识的地方，最终他们结了婚、生了孩子，还合著了这本书。因此，你真的可以责怪 Ousterhout，让你陷入这场混乱。

[P+95]“Informed Prefetching and Caching”
R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka SOSP ’95
关于文件系统中一些非常酷的创意的有趣文章，其中包括应用程序如何向操作系统提供关于它正在访问哪些文件，以及它计划如何访问这些文件的建议。

作业

程序 mlfq.py 允许你查看本章介绍的 MLFQ 调度程序的行为。详情请参阅 README 文件。

问题

1．只用两个工作和两个队列运行几个随机生成的问题。针对每个工作计算 MLFQ 的执行记录。限制每项作业的长度并关闭 I/O，让你的生活更轻松。

2．如何运行调度程序来重现本章中的每个实例？

3．将如何配置调度程序参数，像轮转调度程序那样工作？

4．设计两个工作的负载和调度程序参数，以便一个工作利用较早的规则 4a 和 4b（用-S 标志打开）来“愚弄”调度程序，在特定的时间间隔内获得 99%的 CPU。

5．给定一个系统，其最高队列中的时间片长度为 10ms，你需要如何频繁地将工作推回到最高优先级级别（带有-B 标志），以保证一个长时间运行（并可能饥饿）的工作得到至少 5%的 CPU？

6．调度中有一个问题，即刚完成 I/O 的作业添加在队列的哪一端。-I 标志改变了这个调度模拟器的这方面行为。尝试一些工作负载，看看你是否能看到这个标志的效果。

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。

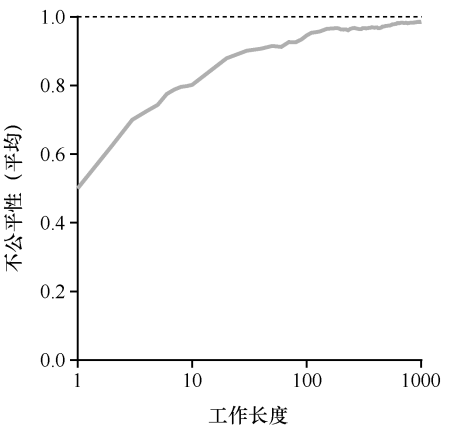

图 9.2 彩票公平性研究

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？