

第 7 章 进程调度：介绍

现在，运行进程的底层机制（mechanism）（如上下文切换）应该清楚了。如果还不清楚，请往回翻一两章，再次阅读这些工作原理的描述。然而，我们还不知道操作系统调度程序采用的上层策略（policy）。接下来会介绍一系列的调度策略（scheduling policy，有时称为 discipline），它们是许多聪明又努力的人在过去这些年里开发的。

事实上，调度的起源早于计算机系统。早期调度策略取自于操作管理领域，并应用于计算机。对于这个事实不必惊讶：装配线以及许多人类活动也需要调度，而且许多关注点是一样的，包括像激光一样清楚的对效率的渴望。因此，我们的问题如下。

关键问题：如何开发调度策略

我们该如何开发一个考虑调度策略的基本框架？什么是关键假设？哪些指标非常重要？哪些基本方法已经在早期的系统中使用？

7.1 工作负载假设

探讨可能的策略范围之前，我们先做一些简化假设。这些假设与系统中运行的进程有关，有时候统称为工作负载（workload）。确定工作负载是构建调度策略的关键部分。工作负载了解得越多，你的策略就越优化。

我们这里做的工作负载的假设是不切实际的，但这没问题（目前），因为我们将来会放宽这些假定，并最终开发出我们所谓的……（戏剧性的暂停）……

一个完全可操作的调度准则（a fully-operational scheduling discipline）^①。

我们对操作系统中运行的进程（有时也叫工作任务）做出如下的假设：

1. 每一个工作运行相同的时间。
2. 所有的工作同时到达。
3. 一旦开始，每个工作保持运行直到完成。
4. 所有的工作只是用 CPU（即它们不执行 IO 操作）。
5. 每个工作的运行时间是已知的。

我们说这些假设中许多是不现实的，但正如在奥威尔的《动物农场》[045]中一些动物比其他动物更平等，本章中的一些假设比其他假设更不现实。特别是，你会很诧异每一个工作的运行时间是已知的——这会让调度程序无所不知，尽管这样很了不起（也许），但最近不太可能发生。

^① 讲这句话的方式和你讲“A fully-operational Death Star.” 的方式一样。

7.2 调度指标

除了做出工作负载假设之外，还需要一个东西能让我们比较不同的调度策略：调度指标。指标是我们用来衡量某些东西的东西，在进程调度中，有一些不同的指标是有意义的。

现在，让我们简化一下生活，只用一个指标：周转时间 (turnaround time)。任务的周转时间定义为任务完成时间减去任务到达系统的时间。更正式的周转时间定义 $T_{\text{周转时间}}$ 是：

$$T_{\text{周转时间}} = T_{\text{完成时间}} - T_{\text{到达时间}} \quad (7.1)$$

因为我们假设所有的任务在同一时间到达，那么 $T_{\text{到达时间}} = 0$ ，因此 $T_{\text{周转时间}} = T_{\text{完成时间}}$ 。随着我们放宽上述假设，这个情况将改变。

你应该注意到，周转时间是一个性能 (performance) 指标，这将是本章的首要关注点。另一个有趣的指标是公平 (fairness)，比如 Jian's Fairness Index[J91]。性能和公平在调度系统中往往是矛盾的。例如，调度程序可以优化性能，但代价是以阻止一些任务运行，这就降低了公平。这个难题也告诉我们，生活并不总是完美的。

7.3 先进先出 (FIFO)

我们可以实现的最基本的算法，被称为先进先出（First In First Out 或 FIFO）调度，有时候也称为先到先服务（First Come First Served 或 FCFS）。

FIFO 有一些积极的特性：它很简单，而且易于实现。而且，对于我们的假设，它的效果很好。

我们一起看一个简单的例子。想象一下，3 个工作 A、B 和 C 在大致相同的时间 ($T_{\text{到达时间}} = 0$) 到达系统。因为 FIFO 必须将某个工作放在前面，所以我们假设当它们都同时到达时，A 比 B 早一点点，然后 B 比 C 早到达一点点。假设每个工作运行 10s。这些工作的平均周转时间 (average turnaround time) 是多少？

从图 7.1 可以看出，A 在 10s 时完成，B 在 20s 时完成，C 在 30s 时完成。因此，这 3 个任务的平均周转时间就是 $(10 + 20 + 30) / 3 = 20$ 。计算周转时间就这么简单。

现在让我们放宽假设。具体来说，让我们放宽假设 1，因此不再认为每个任务的运行时间相同。FIFO 表现如何？你可以构建什么样的工作负载来让 FIFO 表现不好？

（在继续往下读之前，请认真想一下……接着想……想到了么？！）

你现在应该已经弄清楚了，但是以防万一，让我们举个例子来说明不同长度的任务如何导致 FIFO 调度的问题。具体来说，我们再次假设 3 个任务 (A、B 和 C)，但这次 A 运行 100s，而 B 和 C 运行 10s。

如图 7.2 所示，A 先运行 100s，B 或 C 才有机会运行。因此，系统的平均周转时间是比较高的：令人不快的 110s ($(100 + 110 + 120) / 3 = 110$)。

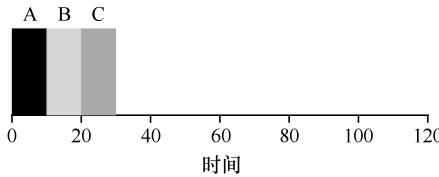


图 7.1 FIFO 的简单例子

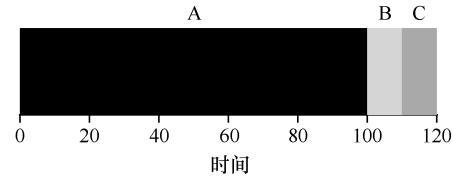


图 7.2 为什么 FIFO 没有那么好

这个问题通常被称为护航效应（convoy effect）[B+79]，一些耗时较少的潜在资源消费者被排在重量级的资源消费者之后。这个调度方案可能让你想起在杂货店只有一个排队队伍的时候，如果看到前面的人装满 3 辆购物车食品并且掏出了支票本，你感觉如何？这会等很长时间^①。

提示：SJF 原则

最短任务优先代表一个总体调度原则，可以应用于所有系统，只要其中平均客户（或在我们案例中的任务）周转时间很重要。想想你等待的任何队伍：如果有关的机构关心客户满意度，他们可能会考虑到 SJF。例如，大超市通常都有一个“零散购物”的通道，以确保仅购买几件东西的购物者，不会堵在为即将到来的冬天而大量购物以做准备的家庭后面。

那么我们该怎么办？如何开发一种更好的算法来处理任务实际运行时间不一样的场景？先考虑一下，然后继续阅读。

7.4 最短任务优先（SJF）

事实证明，一个非常简单的方法解决了这个问题。实际上这是从运筹学中借鉴的一个想法[C54, PV56]，然后应用到计算机系统的任务调度中。这个新的调度准则被称为最短任务优先（Shortest Job First, SJF），该名称应该很容易记住，因为它完全描述了这个策略：先运行最短的任务，然后是次短的任务，如此下去。

我们用上面的例子，但以 SJF 作为调度策略。图 7.3 展示的是运行 A、B 和 C 的结果。它清楚地说明了为什么在考虑平均周转时间的情况下，SJF 调度策略更好。仅通过在 A 之前运行 B 和 C，SJF 将平均周转时间从 110s 降低到 50s ($(10 + 20 + 120) / 3 = 50$)。

事实上，考虑到所有工作同时到达的假设，我们可以证明 SJF 确实是一个最优(optimal)调度算法。但是，你是在上操作系统课，而不是研究理论，所以，这里允许没有证明。

补充：抢占式调度程序

在过去的批处理计算中，开发了一些非抢占式（non-preemptive）调度程序。这样的系统会将每项工作做完，再考虑是否运行新工作。几乎所有现代化的调度程序都是抢占式的（preemptive），非常愿意停止一个进程以运行另一个进程。这意味着调度程序采用了我们之前学习的机制。特别是调度程序可以进行上下文切换，临时停止一个运行进程，并恢复（或启动）另一个进程。

^① 在这种情况下建议采取的措施：要么快速切换到另一个队伍，要么深呼吸并放松。没错，呼气，吸气。这样会变好的，不要担心。

因此，我们找到了一个用 SJF 进行调度的好方法，但是我们的假设仍然是不切实际的。让我们放宽另一个假设。具体来说，我们可以针对假设 2，现在假设工作可以随时到达，而不是同时到达。这导致了什么问题？

(再次停下来想想……你在想吗？加油，你可以做到)

在这里我们可以再次用一个例子来说明问题。现在，假设 A 在 $t=0$ 时到达，且需要运行 100s。而 B 和 C 在 $t=10$ 到达，且各需要运行 10s。用纯 SJF，我们可以得到如图 7.4 所示的调度。

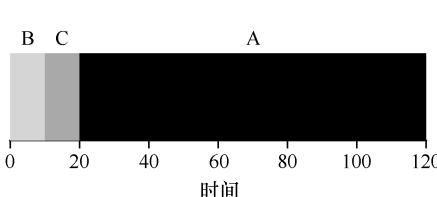


图 7.3 SJF 的简单例子

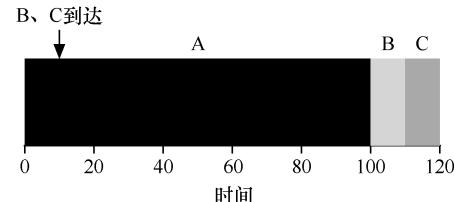


图 7.4 B 和 C 晚到时的 SJF

从图中可以看出，即使 B 和 C 在 A 之后不久到达，它们仍然被迫等到 A 完成，从而遭遇同样的护航问题。这 3 项工作的平均周转时间为 103.33s，即 $(100 + (110 - 10) + (120 - 10)) / 3$ 。

7.5 最短完成时间优先 (STCF)

为了解决这个问题，需要放宽假设条件（工作必须保持运行直到完成）。我们还需要调度程序本身的一些机制。你可能已经猜到，鉴于我们先前关于时钟中断和上下文切换的讨论，当 B 和 C 到达时，调度程序当然可以做其他事情：它可以抢占 (preempt) 工作 A，并决定运行另一个工作，或许稍后继续工作 A。根据我们的定义，SJF 是一种非抢占式 (non-preemptive) 调度程序，因此存在上述问题。

幸运的是，有一个调度程序完全就是这样做的：向 SJF 添加抢占，称为最短完成时间优先 (Shortest Time-to-Completion First, STCF) 或抢占式最短作业优先 (Preemptive Shortest Job First, PSJF) 调度程序[CK68]。每当新工作进入系统时，它就会确定剩余工作和新工作中，谁的剩余时间最少，然后调度该工作。因此，在我们的例子中，STCF 将抢占 A 并运行 B 和 C 以完成。只有在它们完成后，才能调度 A 的剩余时间。图 7.5 展示了一个例子。

结果是平均周转时间大大提高：50s (……)。和以前一样，考虑到我们的新假设，STCF 可证明是最优的。考虑到如果所有工作同时到达，SJF 是最优的，那么你应该能够看到 STCF 的最优性是符合直觉的。

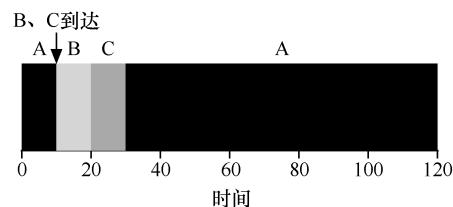


图 7.5 STCF 的简单例子

7.6 新度量指标：响应时间

因此，如果我们知道任务长度，而且任务只使用 CPU，而我们唯一的衡量是周转时间，STCF 将是一个很好的策略。事实上，对于许多早期批处理系统，这些类型的调度算法有一定的意义。然而，引入分时系统改变了这一切。现在，用户将会坐在终端前面，同时也要求系统的交互性好。因此，一个新的度量标准诞生了：响应时间（response time）。

响应时间定义为从任务到达系统到首次运行的时间。更正式的定义是：

$$T_{\text{响应时间}} = T_{\text{首次运行}} - T_{\text{到达时间}} \quad (7.2)$$

例如，如果我们有上面的调度（A 在时间 0 到达，B 和 C 在时间 10 达到），每个作业的响应时间如下：作业 A 为 0，B 为 0，C 为 10（平均：3.33）。

你可能会想，STCF 和相关方法在响应时间上并不是很好。例如，如果 3 个工作同时到达，第三个工作必须等待前两个工作全部运行后才能运行。这种方法虽然有很好的周转时间，但对于响应时间和交互性是相当糟糕的。假设你在终端前输入，不得不等待 10s 才能看到系统的回应，只是因为其他一些工作已经在你之前被调度：你肯定不太开心。

因此，我们还有另一个问题：如何构建对响应时间敏感的调度程序？

7.7 轮转

为了解决这个问题，我们将介绍一种新的调度算法，通常被称为轮转（Round-Robin, RR）调度[K64]。基本思想很简单：RR 在一个时间片（time slice，有时称为调度量子，scheduling quantum）内运行一个工作，然后切换到运行队列中的下一个任务，而不是运行一个任务直到结束。它反复执行，直到所有任务完成。因此，RR 有时被称为时间切片（time-slicing）。请注意，时间片长度必须是时钟中断周期的倍数。因此，如果时钟中断是每 10ms 中断一次，则时间片可以是 10ms、20ms 或 10ms 的任何其他倍数。

为了更详细地理解 RR，我们来看一个例子。假设 3 个任务 A、B 和 C 在系统中同时到达，并且它们都希望运行 5s。SJF 调度程序必须运行完当前任务才可运行下一个任务（见图 7.6）。相比之下，1s 时间片的 RR 可以快速地循环工作（见图 7.7）。

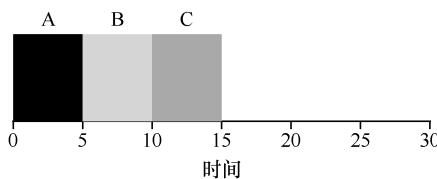


图 7.6 又是 SJF（响应时间不好）

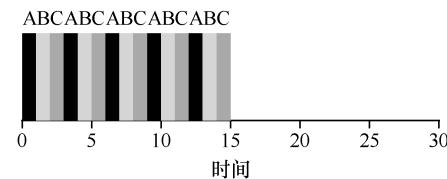


图 7.7 轮转（响应时间好）

RR 的平均响应时间是： $(0 + 1 + 2) / 3 = 1$ ；SJF 算法平均响应时间是： $(0 + 5 + 10) / 3 = 5$ 。如你所见，时间片长度对于 RR 是至关重要的。越短，RR 在响应时间上表现越好。然

而，时间片太短是有问题的：突然上下文切换的成本将影响整体性能。因此，系统设计者需要权衡时间片的长度，使其足够长，以便摊销（amortize）上下文切换成本，而又不会使系统不及时响应。

提示：摊销可以减少成本

当系统某些操作有固定成本时，通常会使用摊销技术（amortization）。通过减少成本的频度（即执行较少次的操作），系统的总成本就会降低。例如，如果时间片设置为 10ms，并且上下文切换时间为 1ms，那么浪费大约 10% 的时间用于上下文切换。如果要摊销这个成本，可以把时间片增加到 100ms。在这种情况下，不到 1% 的时间用于上下文切换，因此时间片带来的成本就被摊销了。

请注意，上下文切换的成本不仅仅来自保存和恢复少量寄存器的操作系统操作。程序运行时，它们在 CPU 高速缓存、TLB、分支预测器和其他片上硬件中建立了大量的状态。切换到另一个工作会导致此状态被刷新，且与当前运行的作业相关的新状态被引入，这可能导致显著的性能成本[MB91]。

如果响应时间是我们的唯一指标，那么带有合理时间片的 RR，就会是非常好的调度程序。但是我们老朋友的周转时间呢？再来看看我们的例子。A、B 和 C，每个运行时间为 5s，同时到达，RR 是具有（长）1s 时间片的调度程序。从图 7.7 可以看出，A 在 13 完成，B 在 14，C 在 15，平均 14。相当可怕！

这并不奇怪，如果周转时间是我们的指标，那么 RR 确实是最糟糕的策略之一。直观地说，这应该是有意义的：RR 所做的正是延伸每个工作，只运行每个工作一小段时间，就转向下一个工作。因为周转时间只关心作业何时完成，RR 几乎是最差的，在很多情况下甚至比简单的 FIFO 更差。

更一般地说，任何公平（fair）的政策（如 RR），即在小规模的时间内将 CPU 均匀分配到活动进程之间，在周转时间这类指标上表现不佳。事实上，这是固有的权衡：如果你愿意不公平，你可以运行较短的工作直到完成，但是要以响应时间为代价。如果你重视公平性，则响应时间会较短，但会以周转时间为代价。这种权衡在系统中很常见。你不能既拥有你的蛋糕，又吃它^①。

我们开发了两种调度程序。第一种类型（SJF、STCF）优化周转时间，但对响应时间不利。第二种类型（RR）优化响应时间，但对周转时间不利。我们还有两个假设需要放宽：假设 4（作业没有 I/O）和假设 5（每个作业的运行时间是已知的）。接下来我们来解决这些假设。

提示：重叠可以提高利用率

如有可能，重叠（overlap）操作可以最大限度地提高系统的利用率。重叠在许多不同的领域很有用，包括执行磁盘 I/O 或将消息发送到远程机器时。在任何一种情况下，开始操作然后切换到其他工作都是一个好主意，这也提高了系统的整体利用率和效率。

^① 这是一个迷惑人的说法，因为它应该是“你不能保留你的蛋糕，又吃它”（这很明显，不是吗？）。令人惊讶的是，这个说法有一个维基百科页面。请自行查阅。

7.8 结合 I/O

首先，我们将放宽假设 4：当然所有程序都执行 I/O。想象一下没有任何输入的程序：每次都会产生相同的输出。设想一个没有输出的程序：它就像谚语所说的森林里倒下的树，没有人看到它。它的运行并不重要。

调度程序显然要在工作发起 I/O 请求时做出决定，因为当前正在运行的作业在 I/O 期间不会使用 CPU，它被阻塞等待 I/O 完成。如果将 I/O 发送到硬盘驱动器，则进程可能会被阻塞几毫秒或更长时间，具体取决于驱动器当前的 I/O 负载。因此，这时调度程序应该在 CPU 上安排另一项工作。

调度程序还必须在 I/O 完成时做出决定。发生这种情况时，会产生中断，操作系统运行并将发出 I/O 的进程从阻塞状态移回就绪状态。当然，它甚至可以决定在那个时候运行该项工作。操作系统应该如何处理每项工作？

为了更好地理解这个问题，让我们假设有两项工作 A 和 B，每项工作需要 50ms 的 CPU 时间。但是，有一个明显的区别：A 运行 10ms，然后发出 I/O 请求（假设 I/O 每个都需要 10ms），而 B 只是使用 CPU 50ms，不执行 I/O。调度程序先运行 A，然后运行 B（见图 7.8）。

假设我们正在尝试构建 STCF 调度程序。这样的调度程序应该如何考虑到这样的事实，即 A 分解成 5 个 10ms 子工作，而 B 仅仅是单个 50ms CPU 需求？显然，仅仅运行一个工作，然后运行另一个工作，而不考虑如何考虑 I/O 是没有意义的。

一种常见的方法是将 A 的每个 10ms 的子工作视为一项独立的工作。因此，当系统启动时，它的选择是调度 10ms 的 A，还是 50ms 的 B。对于 STCF，选择是明确的：选择较短的一个，在这种情况下是 A。然后，A 的工作已完成，只剩下 B，并开始运行。然后提交 A 的一个新子工作，它抢占 B 并运行 10ms。这样做可以实现重叠（overlap），一个进程在等待另一个进程的 I/O 完成时使用 CPU，系统因此得到更好的利用（见图 7.9）。

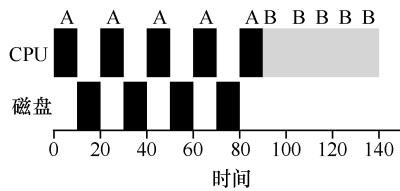


图 7.8 资源的糟糕使用

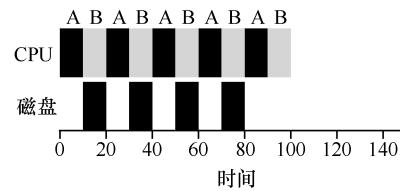


图 7.9 重叠可以更好地使用资源

这样我们就看到了调度程序可能如何结合 I/O。通过将每个 CPU 突发作为一项工作，调度程序确保“交互”的进程经常运行。当这些交互式作业正在执行 I/O 时，其他 CPU 密集型作业将运行，从而更好地利用处理器。

7.9 无法预知

有了应对 I/O 的基本方法，我们来到最后的假设：调度程序知道每个工作的长度。如前

所述，这可能是可以做出的最糟糕的假设。事实上，在一个通用的操作系统中（比如我们所关心的操作系统），操作系统通常对每个作业的长度知之甚少。因此，我们如何建立一个没有这种先验知识的 SJF/STCF？更进一步，我们如何能够将已经看到的一些想法与 RR 调度程序结合起来，以便响应时间也变得相当不错？

7.10 小结

我们介绍了调度的基本思想，并开发了两类方法。第一类是运行最短的工作，从而优化周转时间。第二类是交替运行所有工作，从而优化响应时间。但很难做到“鱼与熊掌兼得”，这是系统中常见的、固有的折中。我们也看到了如何将 I/O 结合到场景中，但仍未解决操作系统根本无法看到未来的问题。稍后，我们将看到如何通过构建一个调度程序，利用最近的历史预测未来，从而解决这个问题。这个调度程序称为多级反馈队列，是第 8 章的主题。

参考资料

[B+79] “The Convoy Phenomenon”

M. Blasgen, J. Gray, M. Mitoma, T. Price

ACM Operating Systems Review, 13:2, April 1979

也许是第一次在数据库和操作系统中提到护航效应。

[C54] “Priority Assignment in Waiting Line Problems”

A. Cobham

Journal of Operations Research, 2:70, pages 70–76, 1954

关于使用 SJF 方法调度修理机器的开创性论文。

[K64] “Analysis of a Time-Shared Processor” Leonard Kleinrock

Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964

该文可能是第一次提到轮转调度算法，当然是调度时分共享系统方法的最早分析之一。

[CK68] “Computer Scheduling Methods and their Countermeasures” Edward G. Coffman and Leonard Kleinrock

AFIPS '68 (Spring), April 1968

一篇很好的早期文章，其中还分析了一些基本调度准则。

[J91] “The Art of Computer Systems Performance Analysis:

Techniques for Experimental Design, Measurement, Simulation, and Modeling”

R. Jain

Interscience, New York, April 1991

计算机系统测量的标准教科书。当然，这对你的库是一个很好的参考。

[PV56] “Machine Repair as a Priority Waiting-Line Problem” Thomas E. Phipps Jr. and W. R. Van Voorhis
Operations Research, 4:1, pages 76–86, February 1956

有关后续工作，概括了来自 Cobham 最初工作的机器修理 SJF 方法，也假定了在这样的环境中 STCF 方法的效用。具体来说，“有一些类型的修理工作，……涉及很多拆卸，地上满是螺母和螺栓，一旦进行就不应该中断。在其他情况下，如果有一个或多个短工作可做，继续做长工作是不可取的（第 81 页）。”

[MB91] “The effect of context switches on cache performance” Jeffrey C. Mogul and Anita Borg
ASPLOS, 1991

关于缓存性能如何受上下文切换影响的一项很好的研究。在今天的系统中问题比较小，如今处理器每秒钟发出数十亿条指令，但上下文切换仍发生在毫秒的时间级别。

作业

`scheduler.py` 这个程序允许你查看不同调度程序在调度指标（如响应时间、周转时间和总等待时间）下的执行情况。详情请参阅 README 文件。

问题

1. 使用 SJF 和 FIFO 调度程序运行长度为 200 的 3 个作业时，计算响应时间和周转时间。
2. 现在做同样的事情，但有不同长度的作业，即 100、200 和 300。
3. 现在做同样的事情，但采用 RR 调度程序，时间片为 1。
4. 对于什么类型的工作负载，SJF 提供与 FIFO 相同的周转时间？
5. 对于什么类型的工作负载和量子长度，SJF 与 RR 提供相同的响应时间？
6. 随着工作长度的增加，SJF 的响应时间会怎样？你能使用模拟程序来展示趋势吗？
7. 随着量子长度的增加，RR 的响应时间会怎样？你能写出一个方程，计算给定 N 个工作时，最坏情况的响应时间吗？