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？

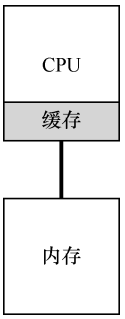


图 10.1 带缓存的单 CPU

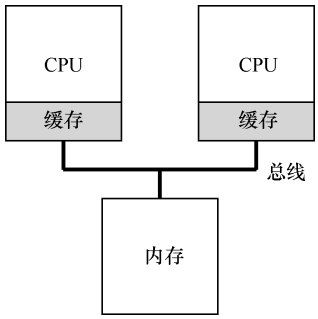


图 10.2 两个有缓存的 CPU 共享内存

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。

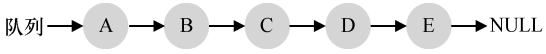
10.4 单队列调度

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）。

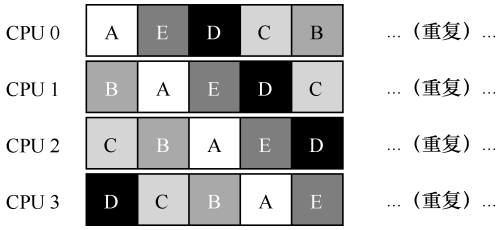
然而，SQMS 有几个明显的短板。第一个是缺乏可扩展性（scalability）。为了保证在多 CPU 上正常运行，调度程序的开发者需要在代码中通过加锁（locking）来保证原子性，如上所述。在 SQMS 访问单个队列时（如寻找下一个运行的工作），锁确保得到正确的结果。

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：



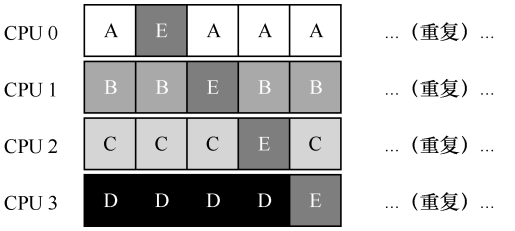
一段时间后，假设每个工作依次执行一个时间片，然后选择另一个工作，下面是每个 CPU 可能的调度序列：



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。

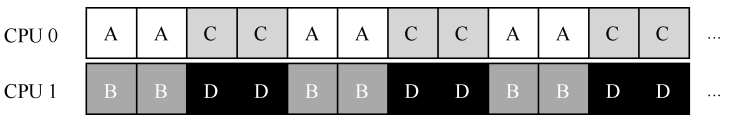
10.5 多队列调度

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：

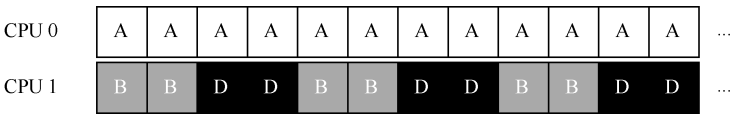


MQMS 比 SQMS 有明显的优势，它天生更具有可扩展性。队列的数量会随着 CPU 的增加而增加，因此锁和缓存争用的开销不是大问题。此外，MQMS 天生具有良好的缓存亲和度。所有工作都保持在固定的 CPU 上，因而可以很好地利用缓存数据。

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

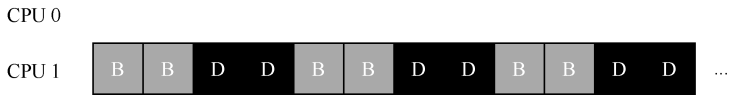
作（如 C）这时执行完毕。现在调度队列如下：

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？

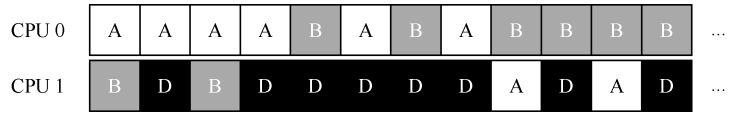
最明显的答案是让工作移动，这种技术我们称为迁移（migration）。通过工作的跨 CPU 迁移，可以真正实现负载均衡。

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。

第 11 章　关于 CPU 虚拟化的总结对话

教授：那么，同学，你学到了什么？

学生：教授，这似乎是一个既定答案的问题。我想你只想让我说“是的，我学到了”。

教授：确实。但这也还是一个诚实的问题。来吧，让教授休息一下，好吗？

学生：好的，好的。我想我确实学到了一些知识。首先，我了解了操作系统如何虚拟化 CPU。为了理解这一点，我必须了解一些重要的机制（mechanism）：陷阱和陷阱处理程序，时钟中断以及操作系统和硬件在进程间切换时如何谨慎地保存和恢复状态。

教授：很好，很好！

学生：虽然所有这些交互似乎有点复杂，但我怎样才能学到更多的内容？

教授：好的，这是一个很好的问题。我认为没有办法可以替代动手。仅阅读这些内容并不能给你正确的理解。做课堂项目，我敢保证，它会对你有所帮助。

学生：听起来不错。我还能告诉你什么？

教授：那么，你在寻求理解操作系统的基本机制时，是否了解了操作系统的哲学？

学生：嗯……我想是的。似乎操作系统相当偏执。它希望确保控制机器。虽然它希望程序能够尽可能高效地运行 [因此也是受限直接执行（limited direct execution）背后的全部逻辑]，但操作系统也希望能够对错误或恶意的程序说“啊！别那么快，我的朋友”。偏执狂全天控制，并且确保操作系统控制机器。也许这就是我们将操作系统视为资源管理器的原因。

教授：是的，听起来你开始融会贯通了！干得漂亮！

学生：谢谢。

教授：那些机制之上的策略呢？有什么有趣的经验吗？

学生：当然能从中学到一些经验。也许有点明显，但明显也可以是很好。比如将短工作提升到队列前面的想法：自从有一次我在商店买一些口香糖，我就知道这是一个好主意，而且我面前的那个人有一张无法支付的信用卡。我要说的是，他不是“短工作”。

教授：这听起来对那个可怜的家伙有点过分。还有什么吗？

学生：好吧，你可以建立一个聪明的调度程序，试图既像 SJF 又像 RR——MLFQ 相当漂亮。构建真正的调度程序似乎很难。

教授：的确如此。这就是对使用哪个调度程序至今仍有争议的原因。例如，请参阅 CFS、BFS 和 O（1）调度程序之间的 Linux 战斗。不，我不会说出 BFS 的全名。

学生：我不会要求你说！这些策略战争看起来好像可以永远持续下去，真的有一个正确的答案吗？

教授：可能没有。毕竟，即使我们自己的度量指标也不一致。如果你的调度程序周转时间好，那么在响应时间就会很糟糕，反之亦然。正如 Lampson 说的，也许目标不是找到最好的解决方案，而是为了避免灾难。

学生：这有点令人沮丧。

教授：好的工程可以这样。它也可以令人振奋！这只是你的观点，真的。我个人认为，务实是一件好事，实用主义者意识到并非所有问题都有简洁明了的解决方案。你还喜欢什么？

学生：我非常喜欢操控调度程序的概念。我下次在亚马逊的 EC2 服务上运行一项工作时，看起来这可能是需要考虑的事情。也许我可以从其他一些毫无戒心的（更重要的是，对操作系统一无所知的）客户那里窃取一些时间周期！

教授：看起来我可能创造了一个“怪物”！你知道，我可不想被人称为弗兰肯斯坦教授。

学生：但你不就是这样想的吗？让我们对某件事感到兴奋，这样我们就会自己对它进行研究？点燃火，仅此而已？

教授：我想是的。但我不认为这会成功！

第 12 章　关于内存虚拟化的对话

学生：那么，虚拟化讲完了吗？

教授：没有！

学生：嘿，没理由这么激动，我只是在问一个问题。学生就应该问问题，对吧？

教授：好吧，教授们总是这样说，但实际上他们的意思：提出问题，仅当它们是好问题，而且你实际上已经对这些问题进行了一些思考。

学生：好吧，那肯定会让我失去动力。

教授：我得逞了。不管怎么说，我们离讲完虚拟化还有一段时间！相反，你刚看到了如何虚拟化 CPU，但是真的有一个巨大的“怪物”—— 内存在壁橱里等着你。虚拟内存很复杂，需要我们理解关于硬件和操作系统交互方式的更多复杂细节。

学生：听起来很酷。为什么这很难？

教授：好吧，有很多细节，你必须牢记它们，才能真正对发生的事情建立一个思维模型。我们将从简单的开始，使用诸如基址/界限等非常基本的技术，并慢慢增加复杂性以应对新的挑战，包括有趣的主题，如 TLB 和多级页表。最终，我们将能够描述一个全功能的现代虚拟内存管理程序的工作原理。

学生：漂亮！对我这个可怜的学生有什么提示吗？会被这些信息淹没，并且一般都会睡眠不足？

教授：对于睡眠不足的人来说，这很简单：多睡一会儿（少一点派对）。对于理解虚拟内存，从这里开始：用户程序生成的每个地址都是虚拟地址（every address generated by a user program is a virtual address）。操作系统只是为每个进程提供一个假象，具体来说，就是它拥有自己的大量私有内存。在一些硬件帮助下，操作系统会将这些假的虚拟地址变成真实的物理地址，从而能够找到想要的信息。

学生：好的，我想我可以记住……（自言自语）用户程序中的每个地址都是虚拟的，用户程序中的每个地址都是虚拟的，每个地址都是……

教授：你在嘟囔什么？

学生：哦，没什么……（尴尬的停顿）……但是，操作系统为什么又要提供这种假象？

教授：主要是为了易于使用（ease of use）。操作系统会让每个程序觉得，它有一个很大的连续地址空间（address space）来放入其代码和数据。因此，作为一名程序员，您不必担心诸如“我应该在哪里存储这个变量？”这样的事情，因为程序的虚拟地址空间很大，有很多空间可以存代码和数据。对于程序员来说，如果必须操心将所有的代码数据放入一个小而拥挤的内存，那么生活会变得痛苦得多。

学生：为什么呢？

教授：好吧，隔离（isolation）和保护（protection）也是大事。我们不希望一个错误的程序能够读取或者覆写其他程序的内存，对吗？

学生：可能不希望。除非它是由你不喜欢的人编写的程序。

教授：嗯……我想可能需要在下个学期为你安排一门道德与伦理课程。也许操作系统课程没有传递正确的信息。

学生：也许应该。但请记住，不是我对大家说，对于错误的进程行为，正确的操作系统反应是要“杀死”违规进程！

第 13 章　抽象：地址空间

早期，构建计算机操作系统非常简单。你可能会问，为什么？因为用户对操作系统的期望不高。然而一些烦人的用户提出要“易于使用”“高性能”“可靠性”等，这导致了所有这些令人头痛的问题。下次你见到这些用户的时候，应该感谢他们，他们是这些问题的根源。

13.1　早期系统

从内存来看，早期的机器并没有提供多少抽象给用户。基本上，机器的物理内存看起来如图 13.1 所示。

操作系统曾经是一组函数（实际上是一个库），在内存中（在本例中，从物理地址 0 开始），然后有一个正在运行的程序（进程），目前在物理内存中（在本例中，从物理地址 64KB 开始），并使用剩余的内存。这里几乎没有抽象，用户对操作系统的要求也不多。那时候，操作系统开发人员的生活确实很容易，不是吗？

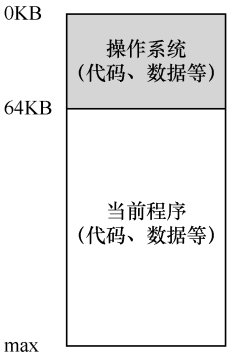


图 13.1　操作系统：早期

13.2　多道程序和时分共享

过了一段时间，由于机器昂贵，人们开始更有效地共享机器。因此，多道程序（multiprogramming）系统时代开启[DV66]，其中多个进程在给定时间准备运行，比如当有一个进程在等待 I/O 操作的时候，操作系统会切换这些进程，这样增加了 CPU 的有效利用率（utilization）。那时候，效率（efficiency）的提高尤其重要，因为每台机器的成本是数十万美元甚至数百万美元（现在你觉得你的 Mac 很贵！）

但很快，人们开始对机器要求更多，分时系统的时代诞生了[S59，L60，M62，M83]。具体来说，许多人意识到批量计算的局限性，尤其是程序员本身[CV65]，他们厌倦了长时间的（因此也是低效率的）编程—调试循环。交互性（interactivity）变得很重要，因为许多用户可能同时在使用机器，每个人都在等待（或希望）他们执行的任务及时响应。

一种实现时分共享的方法，是让一个进程单独占用全部内存运行一小段时间（见图 13.1），然后停止它，并将它所有的状态信息保存在磁盘上（包含所有的物理内存），加载其他进程的状态信息，再运行一段时间，这就实现了某种比较粗糙的机器共享[M+63]。

遗憾的是，这种方法有一个问题：太慢了，特别是当内存增长的时候。虽然保存和恢

复寄存器级的状态信息（程序计数器、通用寄存器等）相对较快，但将全部的内存信息保存到磁盘就太慢了。因此，在进程切换的时候，我们仍然将进程信息放在内存中，这样操作系统可以更有效率地实现时分共享（见图 13.2）。

图 13.2 3 个进程：共享内存

在图 13.2 中，有 3 个进程（A、B、C），每个进程拥有从 512KB 物理内存中切出来给它们的一小部分内存。假定只有一个 CPU，操作系统选择运行其中一个进程（比如 A），同时其他进程（B 和 C）则在队列中等待运行。

随着时分共享变得更流行，人们对操作系统又有了新的要求。特别是多个程序同时驻留在内存中，使保护（protection）成为重要问题。人们不希望一个进程可以读取其他进程的内存，更别说修改了。

13.3 地址空间

然而，我们必须将这些烦人的用户的需求放在心上。因此操作系统需要提供一个易用（easy to use）的物理内存抽象。这个抽象叫作地址空间（address space），是运行的程序看到的系统中的内存。理解这个基本的操作系统内存抽象，是了解内存虚拟化的关键。

一个进程的地址空间包含运行的程序的所有内存状态。比如：程序的代码（code，指令）必须在内存中，因此它们在地址空间里。当程序在运行的时候，利用栈（stack）来保存当前的函数调用信息，分配空间给局部变量，传递参数和函数返回值。最后，堆（heap）用于管理动态分配的、用户管理的内存，就像你从 C 语言中调用 malloc()或面向对象语言（如 C ++或 Java）中调用 new 获得内存。当然，还有其他的东西（例如，静态初始化的变量），但现在假设只有这 3 个部分：代码、栈和堆。

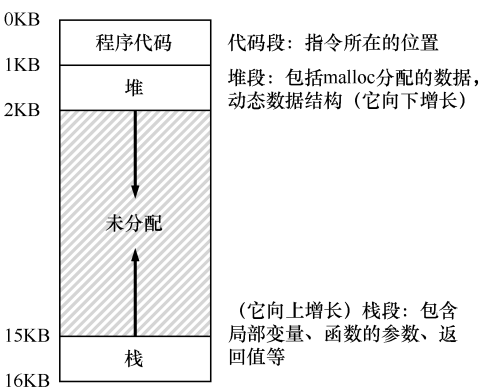


图 13.3 地址空间的例子

在图 13.3 的例子中，我们有一个很小的地址空间[①]（只有 16KB）。程序代码位于地址空间的顶部（在本例中从 0 开始，并且装入到地址空间的前 1KB）。代码是静态的（因此很容易放在内存中），所以可以将它放在地址空间的顶部，我们知道程序运行时不再需要新的空间。

接下来，在程序运行时，地址空间有两个区域可能增长（或者收缩）。它们就是堆（在顶部）和栈（在底部）。把它们放在那里，是因为它们都希望能够增长。通过将它们放在地址空间的两端，我们可以允许这样的增长：它们只需要在相反的方向增长。因此堆在代码（1KB）之下开始并向下增长（当用户通过 malloc()请求更多内存时），栈从 16KB 开始并向上增长

① 我们通常会使用这样的小例子，原因有二：①表示 32 位地址空间是一种痛苦；②数学计算更难。我们喜欢简单的数学。

（当用户进行程序调用时）。然而，堆栈和堆的这种放置方法只是一种约定，如果你愿意，可以用不同的方式安排地址空间 [稍后我们会看到，当多个线程（threads）在地址空间中共存时，就没有像这样分配空间的好办法了]。

当然，当我们描述地址空间时，所描述的是操作系统提供给运行程序的抽象（abstract）。程序不在物理地址 0～16KB 的内存中，而是加载在任意的物理地址。回顾图 13.2 中的进程 A、B 和 C，你可以看到每个进程如何加载到内存中的不同地址。因此问题来了：

关键问题：如何虚拟化内存

操作系统如何在单一的物理内存上为多个运行的进程（所有进程共享内存）构建一个私有的、可能很大的地址空间的抽象？

当操作系统这样做时，我们说操作系统在虚拟化内存（virtualizing memory），因为运行的程序认为它被加载到特定地址（例如 0）的内存中，并且具有非常大的地址空间（例如 32 位或 64 位）。现实很不一样。

例如，当图 13.2 中的进程 A 尝试在地址 0（我们将称其为虚拟地址，virtual address）执行加载操作时，然而操作系统在硬件的支持下，出于某种原因，必须确保不是加载到物理地址 0，而是物理地址 320KB（这是 A 载入内存的地址）。这是内存虚拟化的关键，这是世界上每一个现代计算机系统的基础。

提示：隔离原则

隔离是建立可靠系统的关键原则。如果两个实体相互隔离，这意味着一个实体的失败不会影响另一个实体。操作系统力求让进程彼此隔离，从而防止相互造成伤害。通过内存隔离，操作系统进一步确保运行程序不会影响底层操作系统的操作。一些现代操作系统通过将某些部分与操作系统的其他部分分离，实现进一步的隔离。这样的微内核（microkernel）[BH70，R+89，S+03] 可以比整体内核提供更大的可靠性。

13.4 目标

在这一章中，我们触及操作系统的工作——虚拟化内存。操作系统不仅虚拟化内存，还有一定的风格。为了确保操作系统这样做，我们需要一些目标来指导。以前我们已经看过这些目标（想想本章的前言），我们会再次看到它们，但它们肯定是值得重复的。

虚拟内存（VM）系统的一个主要目标是透明（transparency）[①]。操作系统实现虚拟内存的方式，应该让运行的程序看不见。因此，程序不应该感知到内存被虚拟化的事实，相反，程序的行为就好像它拥有自己的私有物理内存。在幕后，操作系统（和硬件）完成了所有的工作，让不同的工作复用内存，从而实现这个假象。

① 透明的这种用法有时令人困惑。一些学生认为 “变得透明”意味着把所有事情都公之于众。在这里，“变得透明”意味着相反的情况：操作系统提供的假象不应该被应用程序看破。因此，按照通常的用法，透明系统是一个很难注意到的系统。