定证明是更准确的，并且下一个引用也是命中。因此，在我们的简单例子中，LRU 的表现几乎快要赶上最优策略了[①]。

表 22.4 追踪 LUR 策略

访问	命中/未命中	踢出	导致缓存状态	
0	未命中		LUR→	0
1	未命中		LUR→	0、1
2	未命中		LUR→	0、1、2
0	命中		LUR→	1、2、0
1	命中		LUR→	2、0、1
3	未命中	2	LUR→	0、1、3
0	命中		LUR→	1、3、0
3	命中		LUR→	1、0、3
1	命中		LUR→	0、3、1
2	未命中	0	LUR→	3、1、2
1	命中		LUR→	3、2、1

我们也应该注意到，与这些算法完全相反的算法也是存在：最经常使用策略（Most-Frequently-Used，MFU）和最近使用策略（Most-Recently-Used，MRU）。在大多数情况下（不是全部!），这些策略效果都不好，因为它们忽视了大多数程序都具有的局部性特点。

22.6 工作负载示例

让我们再看几个例子，以便更好地理解这些策略。在这里，我们将查看更复杂的工作负载（workload），而不是追踪小例子。但是，这些工作负载也被大大简化了。更好的研究应该包含应用程序追踪。

第一个工作负载没有局部性，这意味着每个引用都是访问一个随机页。在这个简单的例子中，工作负载每次访问独立的 100 个页，随机选择下一个要引用的页。总体来说，访问了 10000 个页。在实验中，我们将缓存大小从非常小（1 页）变化到足以容纳所有页（100 页），以便了解每个策略在缓存大小范围内的表现。

图 22.2 展示了最优、LRU、随机和 FIFO 策略的实验结果。图 22.2 中的 y 轴显示了每个策略的命中率。如上所述，x 轴表示缓存大小的变化。

我们可以从图 22.2 中得出一些结论。首先，当工作负载不存在局部性时，使用的策略区别不大。LRU、FIFO 和随机都执行相同的操作，命中率完全由缓存的大小决定。其次，当缓存足够大到可以容纳所有的数据时，使用哪种策略也无关紧要，所有的策略（甚至是随机的）都有 100%的命中率。最后，你可以看到，最优策略的表现明显好于实际的策略。如果有可能的话，偷窥未来，就能做到更好的替换。

① 好吧，我们夸大了结果。但有时候为了证明一个观点，夸大是有必要的。

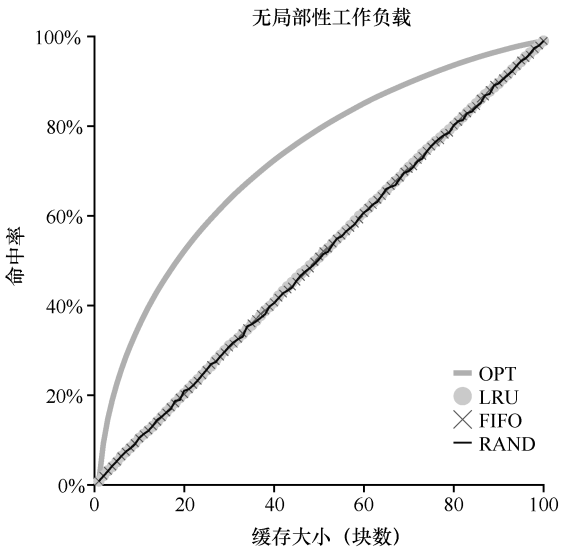

图 22.2 无局部性工作负载

我们下一个工作负载就是所谓的“80—20”负载场景，它表现出局部性：80%的引用是访问 20%的页（“热门”页）。剩下的 20%是对剩余的 80%的页（“冷门”页）访问。在我们的负载场景，总共有 100 个不同的页。因此，“热门”页是大部分时间访问的页，其余时间访问的是“冷门”页。图 22.3 展示了不同策略在这个工作负载下的表现。

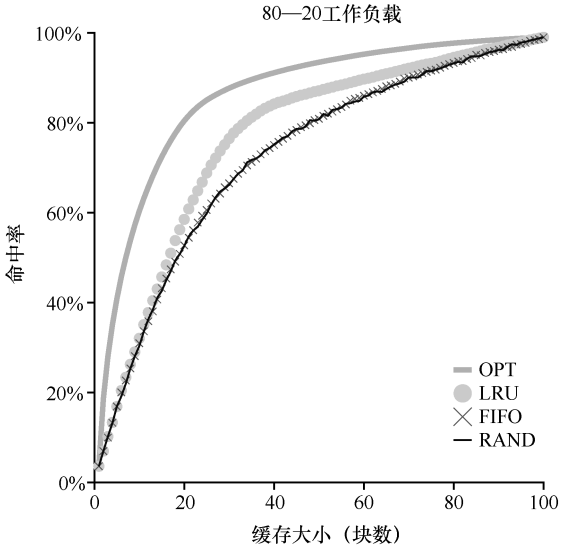

图 22.3 80—20 工作负载

从图 22.3 中可以看出，尽管随机和 FIFO 都很好地运行，但 LRU 更好，因为它更可能保持热门页。由于这些页面过去经常被提及，它们很可能在不久的将来再次被提及。优化再次表现得更好，表明 LRU 的历史信息并不完美。

你现在可能会想：LRU 对随机和 FIFO 的命中率提高真的非常重要么？如往常一样，答案是“视情况而定”。如果每次未命中代价非常大（并不罕见），那么即使小幅提高命中率

（降低未命中率）也会对性能产生巨大的影响。如果未命中的代价不那么大，那么 LRU 带来的好处就不会那么重要。

让我们看看最后一个工作负载。我们称之为“循环顺序”工作负载，其中依次引用 50 个页，从 0 开始，然后是 1，…，49，然后循环，重复访问，总共有 10000 次访问 50 个单独页。图 22.4 展示了这个工作负载下各个策略的行为。

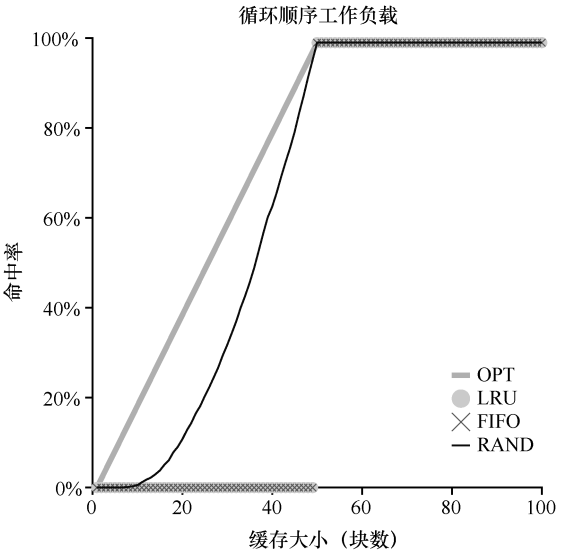

图 22.4 循环工作负载

这种工作负载在许多应用程序（包括重要的商业应用，如数据库[CD85]）中非常常见，展示了 LRU 或者 FIFO 的最差情况。这些算法，在循环顺序的工作负载下，踢出较旧的页。遗憾的是，由于工作负载的循环性质，这些较旧的页将比因为策略决定保存在缓存中的页更早被访问。事实上，即使缓存的大小是 49 页，50 个页面的循环连续工作负载也会导致 0%的命中率。有趣的是，随机策略明显更好，虽然距离最优策略还有距离，但至少达到了非零的命中率。可以看出随机策略有一些不错的属性，比如不会出现特殊情况下奇怪的结果。

22.7 实现基于历史信息的算法

正如你所看到的，像 LRU 这样的算法通常优于简单的策略（如 FIFO 或随机），它们可能会踢出重要的页。遗憾的是，基于历史信息的策略带来了一个新的挑战：应该如何实现呢？

以 LRU 为例。为了实现它，我们需要做很多工作。具体地说，在每次页访问（即每次内存访问，不管是取指令还是加载指令还是存储指令）时，我们都必须更新一些数据，从而将该页移动到列表的前面（即 MRU 侧）。与 FIFO 相比，FIFO 的页列表仅在页被踢出（通过移除最先进入的页）或者当新页添加到列表（已到列表尾部）时才被访问。为了记录哪些页是最少和最近被使用，系统必须对每次内存引用做一些记录工作。显然，如果不十分小心，这样的记录反而会极大地影响性能。

有一种方法有助于加快速度，就是增加一点硬件支持。例如，硬件可以在每个页访问时更新内存中的时间字段（时间字段可以在每个进程的页表中，或者在内存的某个单独的数组中，每个物理页有一个）。因此，当页被访问时，时间字段将被硬件设置为当前时间。然后，在需要替换页时，操作系统可以简单地扫描系统中所有页的时间字段以找到最近最少使用的页。

遗憾的是，随着系统中页数量的增长，扫描所有页的时间字段只是为了找到最精确最少使用的页，这个代价太昂贵。想象一下一台拥有 4GB 内存的机器，内存切成 4KB 的页。这台机器有一百万页，即使以现代 CPU 速度找到 LRU 页也将需要很长时间。这就引出了一个问题：我们是否真的需要找到绝对最旧的页来替换？找到差不多最旧的页可以吗?

关键问题：如何实现 LRU 替换策略

由于实现完美的 LRU 代价非常昂贵，我们能否实现一个近似的 LRU 算法，并且依然能够获得预期的效果?

22.8 近似 LRU

事实证明，答案是肯定的：从计算开销的角度来看，近似 LRU 更为可行，实际上这也是许多现代系统的做法。这个想法需要硬件增加一个使用位（use bit，有时称为引用位，reference bit），这种做法在第一个支持分页的系统 Atlas one-level store[KE + 62]中实现。系统的每个页有一个使用位，然后这些使用位存储在某个地方（例如，它们可能在每个进程的页表中，或者只在某个数组中）。每当页被引用（即读或写）时，硬件将使用位设置为 1。但是，硬件不会清除该位（即将其设置为 0），这由操作系统负责。

操作系统如何利用使用位来实现近似 LRU？可以有很多方法，有一个简单的方法称作时钟算法（clock algorithm）[C69]。想象一下，系统中的所有页都放在一个循环列表中。时钟指针（clock hand）开始时指向某个特定的页（哪个页不重要）。当必须进行页替换时，操作系统检查当前指向的页 P 的使用位是 1 还是 0。如果是 1，则意味着页面 P 最近被使用，因此不适合被替换。然后，P 的使用位设置为 0，时钟指针递增到下一页（$P + 1$）。该算法一直持续到找到一个使用位为 0 的页，使用位为 0 意味着这个页最近没有被使用过（在最坏的情况下，所有的页都已经被使用了，那么就将所有页的使用位都设置为 0）。

请注意，这种方法不是通过使用位来实现近似 LRU 的唯一方法。实际上，任何周期性地清除使用位，然后通过区分使用位是 1 和 0 来判定该替换哪个页的方法都是可以的。Corbato 的时钟算法只是一个早期成熟的算法，并且具有不重复扫描内存来寻找未使用页的特点，也就是它在最差情况下，只会遍历一次所有内存。

图 22.5 展示了时钟算法的一个变种的行为。该变种在需要进行页替换时随机扫描各页，如果遇到一个页的引用位为 1，就清除该位（即将它设置为 0）。直到找到一个使用位为 0 的页，将这个页进行替换。如你所见，虽然时钟算法不如完美的 LRU 做得好，但它比不考虑历史访问的方法要好。

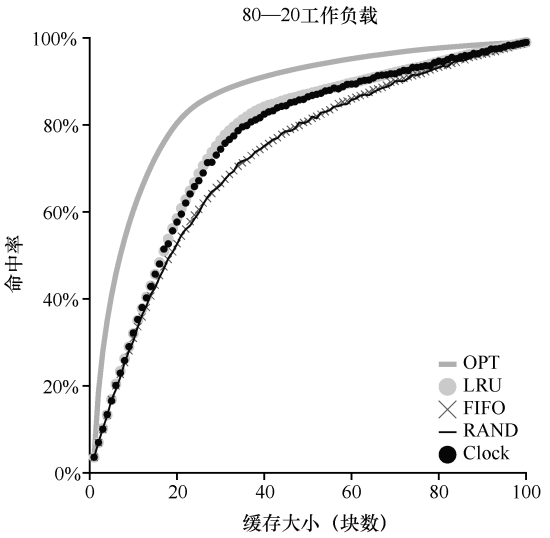

图 22.5 80—20 工作负载和时钟算法

22.9 考虑脏页

时钟算法的一个小修改（最初也由 Corbato [C69]提出），是对内存中的页是否被修改的额外考虑。这样做的原因是：如果页已被修改（modified）并因此变脏（dirty），则踢出它就必须将它写回磁盘，这很昂贵。如果它没有被修改（因此是干净的，clean），踢出就没成本。物理帧可以简单地重用于其他目的而无须额外的 I/O。因此，一些虚拟机系统更倾向于踢出干净页，而不是脏页。

为了支持这种行为，硬件应该包括一个修改位（modified bit，又名脏位，dirty bit）。每次写入页时都会设置此位，因此可以将其合并到页面替换算法中。例如，时钟算法可以被改变，以扫描既未使用又干净的页先踢出。无法找到这种页时，再查找脏的未使用页面，等等。

22.10 其他虚拟内存策略

页面替换不是虚拟内存子系统采用的唯一策略（尽管它可能是最重要的）。例如，操作系统还必须决定何时将页载入内存。该策略有时称为页选择（page selection）策略（因为 Denning 这样命名[D70]），它向操作系统提供了一些不同的选项。

对于大多数页而言，操作系统只是使用按需分页（demand paging），这意味着操作系统在页被访问时将页载入内存中，“按需”即可。当然，操作系统可能会猜测一个页面即将被使用，从而提前载入。这种行为被称为预取（prefetching），只有在有合理的成功机会时才应该这样做。例如，一些系统将假设如果代码页 P 被载入内存，那么代码页 $P+1$ 很可能很快被访问，因此也应该被载入内存。

另一个策略决定了操作系统如何将页面写入磁盘。当然，它们可以简单地一次写出一

个。然而，许多系统会在内存中收集一些待完成写入，并以一种（更高效）的写入方式将它们写入硬盘。这种行为通常称为聚集（clustering）写入，或者就是分组写入（grouping），这样做有效是因为硬盘驱动器的性质，执行单次大的写操作，比许多小的写操作更有效。

22.11 抖动

在结束之前，我们解决了最后一个问题：当内存就是被超额请求时，操作系统应该做什么，这组正在运行的进程的内存需求是否超出了可用物理内存？在这种情况下，系统将不断地进行换页，这种情况有时被称为抖动（thrashing）[D70]。

一些早期的操作系统有一组相当复杂的机制，以便在抖动发生时检测并应对。例如，给定一组进程，系统可以决定不运行部分进程，希望减少的进程工作集（它们活跃使用的页面）能放入内存，从而能够取得进展。这种方法通常被称为准入控制（admission control），它表明，少做工作有时比尝试一下子做好所有事情更好，这是我们在现实生活中以及在现代计算机系统中经常遇到的情况（令人遗憾）。

目前的一些系统采用更严格的方法处理内存过载。例如，当内存超额请求时，某些版本的 Linux 会运行“内存不足的杀手程序（out-of-memory killer）”。这个守护进程选择一个内存密集型进程并杀死它，从而以不怎么委婉的方式减少内存。虽然成功地减轻了内存压力，但这种方法可能会遇到问题，例如，如果它杀死 *X* 服务器，就会导致所有需要显示的应用程序不可用。

22.12 小结

我们已经看到了许多页替换（和其他）策略的介绍，这些策略是所有现代操作系统中虚拟内存子系统的一部分。现代系统增加了对时钟等简单 LRU 近似值的一些调整。例如，扫描抗性（scan resistance）是许多现代算法的重要组成部分，如 ARC [MM03]。扫描抗性算法通常是类似 LRU 的，但也试图避免 LRU 的最坏情况行为，我们曾在循环顺序工作负载中看到这种情况。因此，页替换算法的发展仍在继续。

然而，在许多情况下，由于内存访问和磁盘访问时间之间的差异增加，这些算法的重要性降低了。由于分页到硬盘非常昂贵，因此频繁分页的成本太高。所以，过度分页的最佳解决方案往往很简单：购买更多的内存。

参考资料

[AD03]“Run-Time Adaptation in River”Remzi H. Arpaci-Dusseau

ACM TOCS, 21:1, February 2003

本书作者之一关于 River 系统的研究工作的总结。当然，在其中，他发现与理想情况做比较是系统设计人员的一项重要技术。

[B66]“A Study of Replacement Algorithms for Virtual-Storage Computer”Laszlo A. Belady
IBM Systems Journal 5(2): 78-101, 1966
这篇文章介绍了计算策略最优行为的简单方法（MIN 算法）。

[BNS69]“An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine”
L. A. Belady and R. A. Nelson and G. S. Shedler
Communications of the ACM, 12:6, June 1969
介绍称为“Belady 的异常”的内存引用的小序列的文章。我们想知道，Nelson 和 Shedler 如何看待这个名字呢？

[CD85]“An Evaluation of Buffer Management Strategies for Relational Database Systems”Hong-Tai Chou and David J. DeWitt
VLDB ’85, Stockholm, Sweden, August 1985
关于不同缓冲策略的著名数据库文章，你应该使用多种常见数据库访问模式。如果你知道有关工作负载的某些信息，那么就可以制订策略，让它比操作系统中常见的通用目标策略更好。

[C69]“A Paging Experiment with the Multics System”
F.J. Corbato
Included in a Festschrift published in honor of Prof. P.M. Morse MIT Press, Cambridge, MA, 1969
对时钟算法的最初引用（很难找到!），但不是第一次使用位。感谢麻省理工学院的 H. Balakrishnan 为我们找出这篇论文。

[D70]“Virtual Memory”Peter J. Denning
Computing Surveys, Vol. 2, No. 3, September 1970
Denning 对虚拟存储系统的早期著名调查。

[EF78]“Cold-start vs. Warm-start Miss Ratios”Malcolm C. Easton and Ronald Fagin Communications of the ACM, 21:10, October 1978
关于冷启动与热启动未命中的很好的讨论。

[FP89]“Electrochemically Induced Nuclear Fusion of Deuterium”Martin Fleischmann and Stanley Pons
Journal of Electroanalytical Chemistry, Volume 26, Number 2, Part 1, April, 1989
这篇著名的论文有可能为世界带来革命性的变化，它提供了一种简单的方法，可以从水罐中产生几乎无限的电力，而电力罐中只含有少许金属。但 Pons 和 Fleischmann 发表的（并广为宣传的）实验结果无法重现，因此这两位有梦想的科学家都丧失了名誉（当然也受到了嘲笑）。唯一真正为这个结果感到高兴的人是 Marvin Hawkins，尽管他参与了这项工作，但他的名字却从本文中被删除了。他因而避免将他的名字与 20 世纪最大的科学失误之一联系起来。

[HP06]“Computer Architecture: A Quantitative Approach”John Hennessy and David Patterson
Morgan-Kaufmann, 2006
一本关于计算机体系结构的了不起而奇妙的书，必读！

[H87]“Aspects of Cache Memory and Instruction Buffer Performance”Mark D. Hill

Ph.D. Dissertation, U.C. Berkeley, 1987
Mark Hill 在其论文工作中介绍了 3C，后来因其被包含在 H&P [HP06]中而广泛流行。其中的引述：“我发现根据未命中的原因直观地将未命中划分为 3 个部分是有用的（第 49 页）。”

[KE+62]“One-level Storage System”
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner IRE Trans. EC-11:2, 1962
虽然 Atlas 有一个使用位，但它只有很少量的页，因此在大型存储器中使用位的扫描并不是作者解决的问题。

[M+70]“Evaluation Techniques for Storage Hierarchies”
R．L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger IBM Systems Journal, Volume 9:2, 1970
一篇主要关于如何高效地模拟缓存层次结构的论文。本文无疑是这方面的经典之作，还有对各种替代算法的一些特性的极佳讨论。你能弄清楚为什么栈属性可能对同时模拟很多不同大小的缓存有用吗？

[MM03]“ARC: A Self-Tuning, Low Overhead Replacement Cache”Nimrod Megiddo and Dharmendra S. Modha FAST 2003, February 2003, San Jose, California
关于替换算法的优秀现代论文，其中包括现在某些系统中使用的新策略 ARC。在 2014 年 FAST ’14 大会上，获得了存储系统社区的“时间考验”奖。

作业

这个模拟器 paging-policy.py 允许你使用不同的页替换策略。详情请参阅 README 文件。

问题

1．使用以下参数生成随机地址：-s 0 -n 10，-s 1 -n 10 和-s 2 -n 10。将策略从 FIFO 更改为 LRU，并将其更改为 OPT。计算所述地址追踪中的每个访问是否命中或未命中。

2．对于大小为 5 的高速缓存，为以下每个策略生成最差情况的地址引用序列：FIFO、LRU 和 MRU（最差情况下的引用序列导致尽可能多的未命中）。对于最差情况下的引用序列，需要的缓存增大多少，才能大幅提高性能，并接近 OPT？

3．生成一个随机追踪序列（使用 Python 或 Perl）。你预计不同的策略在这样的追踪序列上的表现如何？

4．现在生成一些局部性追踪序列。如何能够产生这样的追踪序列？LRU 表现如何？RAND 比 LRU 好多少？CLOCK 表现如何？CLOCK 使用不同数量的时钟位，表现如何？

5．使用像 valgrind 这样的程序来测试真实应用程序并生成虚拟页面引用序列。例如，运行 valgrind --tool = lackey --trace-mem = yes ls 将为程序 ls 所做的每个指令和数据引用，输出近乎完整的引用追踪。为了使上述仿真器有用，你必须首先将每个虚拟内存引用转换为虚拟页码参考（通过屏蔽偏移量并向右移位来完成）。为了满足大部分请求，你的应用程序追踪需要多大的缓存？随着缓存大小的增加绘制其工作集的图形。

第 23 章　VAX/VMS 虚拟内存系统

在我们结束对虚拟内存的研究之前，让我们仔细研究一下 VAX/VMS 操作系统[LL82]的虚拟内存管理器，它特别干净漂亮。本章将讨论该系统，说明如何在一个完整的内存管理器中，将先前章节中提出的一些概念结合在一起。

23.1　背景

数字设备公司（DEC）在 20 世纪 70 年代末推出了 VAX-11 小型机体系结构。在微型计算机时代，DEC 是计算机行业的一个大玩家。遗憾的是，一系列糟糕的决定和个人计算机的出现慢慢（但不可避免地）导致该公司走向倒闭[C03]。该架构有许多实现，包括 VAX-11/780 和功能较弱的 VAX-11/750。

该系统的操作系统被称为 VAX/VMS（或者简单的 VMS），其主要架构师之一是 Dave Cutler，他后来领导开发了微软 Windows NT [C93]。VMS 面临通用性的问题，即它将运行在各种机器上，包括非常便宜的 VAXen（是的，这是正确的复数形式），以及同一架构系列中极高端和强大的机器。因此，操作系统必须具有一些机制和策略，适用于这一系列广泛的系统（并且运行良好）。

关键问题：如何避免通用性“魔咒”

操作系统常常有所谓的“通用性魔咒”问题，它们的任务是为广泛的应用程序和系统提供一般支持。其根本结果是操作系统不太可能很好地支持任何一个安装。VAX-11 体系结构有许多不同的实现。那么，如何构建操作系统以便在各种系统上有效运行？

附带说一句，VMS 是软件创新的很好例子，用于隐藏架构的一些固有缺陷。尽管操作系统通常依靠硬件来构建高效的抽象和假象，但有时硬件设计人员并没有把所有事情都做好。在 VAX 硬件中，我们会看到一些例子，也会看到尽管存在这些硬件缺陷，VMS 操作系统如何构建一个有效的工作系统。

23.2　内存管理硬件

VAX-11 为每个进程提供了一个 32 位的虚拟地址空间，分为 512 字节的页。因此，虚拟地址由 23 位 VPN 和 9 位偏移组成。此外，VPN 的高两位用于区分页所在的段。因此，如前所述，该系统是分页和分段的混合体。

地址空间的下半部分称为“进程空间”，对于每个进程都是唯一的。在进程空间的前半部分（称为 P0）中，有用户程序和一个向下增长的堆。在进程空间的后半部分（P1），有向上增长的栈。地址空间的上半部分称为系统空间（S），尽管只有一半被使用。受保护的操作系统代码和数据驻留在此处，操作系统以这种方式跨进程共享。

VMS 设计人员的一个主要关注点是 VAX 硬件中的页大小非常小（512 字节）。由于历史原因选择的这种尺寸，存在一个根本性问题，即简单的线性页表过大。因此，VMS 设计人员的首要目标之一是确保 VMS 不会用页表占满内存。

系统通过两种方式，减少了页表对内存的压力。首先，通过将用户地址空间分成两部分，VAX-11 为每个进程的每个区域（P0 和 P1）提供了一个页表。因此，栈和堆之间未使用的地址空间部分不需要页表空间。基址和界限寄存器的使用与你期望的一样。一个基址寄存器保存该段的页表的地址，界限寄存器保存其大小（即页表项的数量）。

其次，通过在内核虚拟内存中放置用户页表（对于 P0 和 P1，因此每个进程两个），操作系统进一步降低了内存压力。因此，在分配或增长页表时，内核在段 S 中分配自己的虚拟内存空间。如果内存受到严重压力，内核可以将这些页表的页面交换到磁盘，从而使物理内存可以用于其他用途。

将页表放入内核虚拟内存意味着地址转换更加复杂。例如，要转换 P0 或 P1 中的虚拟地址，硬件必须首先尝试在其页表中查找该页的页表项（该进程的 P0 或 P1 页表）。但是，在这样做时，硬件可能首先需要查阅系统页表（它存在于物理内存中）。随着地址转换完成，硬件可以知道页表页的地址，然后最终知道所需内存访问的地址。幸运的是，VAX 的硬件管理的 TLB 让所有这些工作更快，TLB 通常（很有可能）会绕过这种费力的查找。

23.3 一个真实的地址空间

研究 VMS 有一个很好的方面，我们可以看到如何构建一个真正的地址空间（见图 23.1）。到目前为止，我们一直假设了一个简单的地址空间，只有用户代码、用户数据和用户堆，但正如我们上面所看到的，真正的地址空间显然更复杂。

补充：为什么空指针访问会导致段错误

你现在应该很好地理解一个空指针引用会发生什么。通过这样做，进程生成了一个虚拟地址 0：

```
int *p = NULL; // set p = 0
*p = 10;       // try to store value 10 to virtual address 0
```

硬件试图在 TLB 中查找 VPN（这里也是 0），遇到 TLB 未命中。查询页表，并且发现 VPN 0 的条目被标记为无效。因此，我们遇到无效的访问，将控制权交给操作系统，这可能会终止进程（在 UNIX 系统上，会向进程发出一个信号，让它们对这样的错误做出反应。但是如果信号未被捕获，则会终止进程）。

例如，代码段永远不会从第 0 页开始。相反，该页被标记为不可访问，以便为检测空指针（null-pointer）访问提供一些支持。因此，设计地址空间时需要考虑的一个问题是对调试的支持，这正是无法访问的零页所提供的。

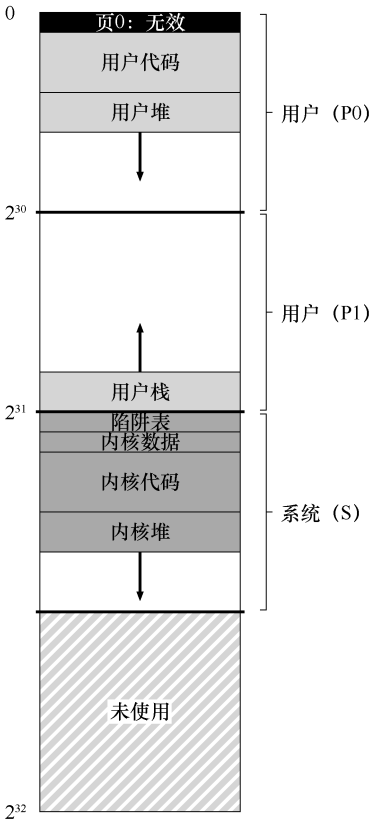

图 23.1 VAX / VMS 地址空间

也许更重要的是，内核虚拟地址空间（即其数据结构和代码）是每个用户地址空间的一部分。在上下文切换时，操作系统改变 P0 和 P1 寄存器以指向即将运行的进程的适当页表。但是，它不会更改 S 基址和界限寄存器，并因此将“相同的”内核结构映射到每个用户的地址空间。

内核映射到每个地址空间，这有一些原因。这种结构使得内核的运转更轻松。例如，如果操作系统收到用户程序（例如，在 write()系统调用中）递交的指针，很容易将数据从该指针处复制到它自己的结构。操作系统自然是写好和编译好的，无须担心它访问的数据来自哪里。相反，如果内核完全位于物理内存中，那么将页表的交换页切换到磁盘是非常困难的。如果内核被赋予了自己的地址空间，那么在用户应用程序和内核之间移动数据将再次变得复杂和痛苦。通过这种构造（现在广泛使用），内核几乎就像应用程序库一样，尽管是受保护的。

关于这个地址空间的最后一点与保护有关。显然，操作系统不希望用户应用程序读取或写入操作系统数据或代码。因此，硬件必须支持页面的不同保护级别才能启用该功能。VAX 通过在页表中的保护位中指定 CPU 访问特定页面所需的特权级别来实现此目的。因此，系统数据和代码被设置为比用户数据和代码更高的保护级别。试图从用户代码访问这些信息，将会在操作系统中产生一个陷阱，并且（你猜对了）可能会终止违规进程。

23.4 页替换

VAX 中的页表项（PTE）包含以下位：一个有效位，一个保护字段（4 位），一个修改（或脏位）位，为 OS 使用保留的字段（5 位），最后是一个物理帧号码（PFN）将页面的位置存储在物理内存中。敏锐的读者可能会注意到：没有引用位（no reference bit）！因此，VMS 替换算法必须在没有硬件支持的情况下，确定哪些页是活跃的。

开发人员也担心会有“自私贪婪的内存”（memory hog）——一些程序占用大量内存，使其他程序难以运行。到目前为止，我们所看到的大部分策略都容易受到这种内存的影响。例如，LRU 是一种全局策略，不会在进程之间公平分享内存。

分段的 FIFO

为了解决这两个问题，开发人员提出了分段的 FIFO（segmented FIFO）替换策略[RL81]。想法很简单：每个进程都有一个可以保存在内存中的最大页数，称为驻留集大小（Resident Set Size，RSS）。每个页都保存在 FIFO 列表中。当一个进程超过其 RSS 时，“先入”的页被驱逐。FIFO 显然不需要硬件的任何支持，因此很容易实现。

正如我们前面看到的，纯粹的 FIFO 并不是特别好。为了提高 FIFO 的性能，VMS 引入了两个二次机会列表（second-chance list），页在从内存中被踢出之前被放在其中。具体来说，是全局的干净页空闲列表和脏页列表。当进程 P 超过其 RSS 时，将从其每个进程的 FIFO 中移除一个页。如果干净（未修改），则将其放在干净页列表的末尾。如果脏（已修改），则将其放在脏页列表的末尾。

如果另一个进程 Q 需要一个空闲页，它会从全局干净列表中取出第一个空闲页。但是，如果原来的进程 P 在回收之前在该页上出现页错误，则 P 会从空闲（或脏）列表中回收，从而避免昂贵的磁盘访问。这些全局二次机会列表越大，分段的 FIFO 算法越接近 LRU [RL81]。

页聚集

VMS 采用的另一个优化也有助于克服 VMS 中的小页面问题。具体来说，对于这样的小页面，交换过程中的硬盘 I/O 可能效率非常低，因为硬盘在大型传输中效果更好。为了让交换 I/O 更有效，VMS 增加了一些优化，但最重要的是聚集（clustering）。通过聚集，VMS 将大批量的页从全局脏列表中分组到一起，并将它们一举写入磁盘（从而使它们变干净）。聚集用于大多数现代系统，因为可以在交换空间的任意位置放置页，所以操作系统对页分组，执行更少和更大的写入，从而提高性能。

补充：模拟引用位

事实证明，你不需要硬件引用位，就可以了解系统中哪些页在用。事实上，在 20 世纪 80 年代早期，Babaoglu 和 Joy 表明，VAX 上的保护位可以用来模拟引用位[BJ81]。其基本思路是：如果你想了解哪些页在系统中被活跃使用，请将页表中的所有页标记为不可访问（但请注意关于哪些页可以被进程真正访问

的信息，也许在页表项的“保留的操作系统字段”部分）。当一个进程访问一页时，它会在操作系统中产生一个陷阱。操作系统将检查页是否真的可以访问，如果是，则将该页恢复为正常保护（例如，只读或读写）。在替换时，操作系统可以检查哪些页仍然标记为不可用，从而了解哪些页最近没有被使用过。

这种引用位“模拟”的关键是减少开销，同时仍能很好地了解页的使用。标记页不可访问时，操作系统不应太激进，否则开销会过高。同时，操作系统也不能太被动，否则所有页面都会被引用，操作系统又无法知道踢出哪一页。

23.5 其他漂亮的虚拟内存技巧

VMS 有另外两个现在成为标准的技巧：按需置零和写入时复制。我们现在描述这些惰性（lazy）优化。

VMS（以及大多数现代系统）中的一种懒惰形式是页的按需置零（demand zeroing）。为了更好地理解这一点，我们来考虑一下在你的地址空间中添加一个页的例子。在一个初级实现中，操作系统响应一个请求，在物理内存中找到页，将该页添加到你的堆中，并将其置零（安全起见，这是必需的。否则，你可以看到其他进程使用该页时的内容。），然后将其映射到你的地址空间（设置页表以根据需要引用该物理页）。但是初级实现可能是昂贵的，特别是如果页没有被进程使用。

利用按需置零，当页添加到你的地址空间时，操作系统的工作很少。它会在页表中放入一个标记页不可访问的条目。如果进程读取或写入页，则会向操作系统发送陷阱。在处理陷阱时，操作系统注意到（通常通过页表项中“保留的操作系统字段”部分标记的一些位），这实际上是一个按需置零页。此时，操作系统会完成寻找物理页的必要工作，将它置零，并映射到进程的地址空间。如果该进程从不访问该页，则所有这些工作都可以避免，从而体现按需置零的好处。

提示：惰性

惰性可以使得工作推迟，但出于多种原因，这在操作系统中是有益的。首先，推迟工作可能会减少当前操作的延迟，从而提高响应能力。例如，操作系统通常会报告立即写入文件成功，只是稍后在后台将其写入硬盘。其次，更重要的是，惰性有时会完全避免完成这项工作。例如，延迟写入直到文件被删除，根本不需要写入。

VMS 有另一个很酷的优化（几乎每个现代操作系统都是这样），写时复制（copy-on-write，COW）。这个想法至少可以回溯到 TENEX 操作系统[BB+72]，它很简单：如果操作系统需要将一个页面从一个地址空间复制到另一个地址空间，不是实际复制它，而是将其映射到目标地址空间，并在两个地址空间中将其标记为只读。如果两个地址空间都只读取页面，则不会采取进一步的操作，因此操作系统已经实现了快速复制而不实际移动任何数据。

但是，如果其中一个地址空间确实尝试写入页面，就会陷入操作系统。操作系统会注意到该页面是一个 COW 页面，因此（惰性地）分配一个新页，填充数据，并将这个新页映

射到错误处理的地址空间。该进程然后继续，现在有了该页的私人副本。

COW 有用有一些原因。当然，任何类型的共享库都可以通过写时复制，映射到许多进程的地址空间中，从而节省宝贵的内存空间。在 UNIX 系统中，由于 fork()和 exec()的语义，COW 更加关键。你可能还记得，fork()会创建调用者地址空间的精确副本。对于大的地址空间，这样的复制过程很慢，并且是数据密集的。更糟糕的是，大部分地址空间会被随后的 exec()调用立即覆盖，它用即将执行的程序覆盖调用进程的地址空间。通过改为执行写时复制的 fork()，操作系统避免了大量不必要的复制，从而保留了正确的语义，同时提高了性能。

23.6 小结

现在我们已经从头到尾地复习整个虚拟存储系统。希望大多数细节都很容易明白，因为你应该已经对大部分基本机制和策略有了很好的理解。Levy 和 Lipman [LL82]出色的（简短的）论文中有更详细的介绍。建议你阅读它，这是了解这些章节背后的资源来源的好方法。

在可能的情况下，你还应该通过阅读 Linux 和其他现代系统来了解更多关于最新技术的信息。有很多原始资料，包括一些不错的书籍[BC05]。有一件事会让你感到惊讶：在诸如 VAX/VMS 这样的较早论文中看到的经典理念，仍然影响着现代操作系统的构建方式。

参考资料

[BB+72]“TENEX, A Paged Time Sharing System for the PDP-10”
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson Communications of the ACM, Volume 15, March 1972
早期的分时操作系统，有许多好的想法来自于此。写时复制只是其中之一，在这里可以找到现代系统许多其他方面的灵感，包括进程管理、虚拟内存和文件系统。

[BJ81]“Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits”Ozalp Babaoglu and William N. Joy
SOSP ’81, Pacific Grove, California, December 1981
关于如何利用机器内现有的保护机制来模拟引用位的巧妙构思。这个想法来自 Berkeley 的小组，他们正在致力于开发他们自己的 UNIX 版本，也就是所谓的伯克利系统发行版，或称为 BSD。该团队在 UNIX 的发展中有很大的影响力，包括虚拟内存、文件系统和网络方面。

[BC05]“Understanding the Linux Kernel (Third Edition)”Daniel P. Bovet and Marco Cesati
O’Reilly Media, November 2005
关于 Linux 的众多图书之一。它们很快就会过时，但许多基础知识仍然存在，值得一读。

[C03]“The Innovator’s Dilemma”Clayton M. Christenson
Harper Paperbacks, January 2003
一本关于硬盘驱动器行业的精彩图书，还涉及新的创新如何颠覆现有的技术。对于商科和计算机科学家来说，这是一本好书，提供了关于大型成功的公司如何完全失败的洞见。

[C93]“Inside Windows NT”Helen Custer and David Solomon Microsoft Press, 1993
这本关于 Windows NT 的书从头到尾地解释了系统，内容过于详细，你可能不喜欢。但认真地说，它是一本相当不错的书。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Henry M. Levy, Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
这是本章的大部分原始材料来源，它简洁易读。尤其重要的是，如果你想读研究生，你需要做的就是阅读论文、工作，阅读更多的论文、做更多的工作，最后写一篇论文，然后继续工作。但它很有趣！

[RL81]“Segmented FIFO Page Replacement”Rollins Turner and Henry Levy
SIGMETRICS ’81, Las Vegas, Nevada, September 1981
一篇简短的文章，显示了对于一些工作负载，分段的 FIFO 可以接近 LRU 的性能。