程一个）。页表项告诉我们哪个进程正在使用此页，以及该进程的哪个虚拟页映射到此物理页。

现在，要找到正确的项，就是要搜索这个数据结构。线性扫描是昂贵的，因此通常在此基础结构上建立散列表，以加速查找。PowerPC 就是这种架构[JM98]的一个例子。

更一般地说，反向页表说明了我们从一开始就说过的内容：页表只是数据结构。你可以对数据结构做很多疯狂的事情，让它们更小或更大，使它们变得更慢或更快。多层和反向页表只是人们可以做的很多事情的两个例子。

20.5 将页表交换到磁盘

最后，我们讨论放松最后一个假设。到目前为止，我们一直假设页表位于内核拥有的物理内存中。即使我们有很多技巧来减小页表的大小，但是它仍然有可能是太大而无法一次装入内存。因此，一些系统将这样的页表放入内核虚拟内存（kernel virtual memory），从而允许系统在内存压力较大时，将这些页表中的一部分交换（swap）到磁盘。我们将在下一章（即 VAX/VMS 的案例研究）中进一步讨论这个问题，在我们更详细地了解了如何将页移入和移出内存之后。

20.6 小结

我们现在已经看到了如何构建真正的页表。不一定只是线性数组，而是更复杂的数据结构。这样的页表体现了时间和空间上的折中（表格越大，TLB 未命中可以处理得更快，反之亦然），因此结构的正确选择强烈依赖于给定环境的约束。

在一个内存受限的系统中（像很多旧系统一样），小结构是有意义的。在具有较多内存，并且工作负载主动使用大量内存页的系统中，用更大的页表来加速 TLB 未命中处理，可能是正确的选择。有了软件管理的 TLB，数据结构的整个世界开放给了喜悦的操作系统创新者（提示：就是你）。你能想出什么样的新结构？它们解决了什么问题？当你入睡时想想这些问题，做一个只有操作系统开发人员才能做的大梦。

参考资料

[BOH10]“Computer Systems: A Programmer’s Perspective”Randal E. Bryant and David R. O’Hallaron
Addison-Wesley, 2010
我们还没有找到很好的多级页表首选参考。然而，Bryant 和 O’Hallaron 编写的这本了不起的教科书深入探讨了 x86 的细节，至少这是一个使用这种结构的早期系统。这也是一本很棒的书。

[JM98]“Virtual Memory: Issues of Implementation”Bruce Jacob and Trevor Mudge
IEEE Computer, June 1998

对许多不同系统及其虚拟内存方法的优秀调查。其中有关于 x86、PowerPC、MIPS 和其他体系结构的大量细节内容。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
一篇关于经典操作系统 VMS 中真实虚拟内存管理程序的精彩论文。它非常棒，实际上，从现在开始的几章，我们将利用它来复习目前为止我们学过的有关虚拟内存的所有内容。

[M28]“Reese’s Peanut Butter Cups”Mars Candy Corporation.
显然，这些精美的“甜点”是由 Harry Burnett Reese 在 1928 年发明的，他以前曾是奶牛场的农夫和 Milton S. Hershey 的运输工长。至少，维基百科上是这么说的。

[N+02]“Practical, Transparent Operating System Support for Superpages”Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox
OSDI ’02, Boston, Massachusetts, October 2002
一篇精彩的论文，展示了将大页或超大页并入现代操作系统中的所有细节。这篇文章阅读起来没有你想象的那么容易。

[M07]“Multics: History”
这个神奇的网站提供了 Multics 系统的大量历史记录，当然是 OS 历史上最有影响力的系统之一。引文如下：“麻省理工学院的 Jack Dennis 为 Multics 的开始提供了有影响力的架构理念，特别是将分页和分段相结合的想法。”

作业

这个有趣的小作业会测试你是否了解多级页表的工作原理。是的，前面句子中使用的“有趣”一词有一些争议。该程序叫作“可能不太怪：paging-multilevel-translate.py”。详情请参阅 README 文件。

问题

1. 对于线性页表，你需要一个寄存器来定位页表，假设硬件在 TLB 未命中时进行查找。你需要多少个寄存器才能找到两级页表？三级页表呢？

2. 使用模拟器对随机种子 0、1 和 2 执行翻译，并使用-c 标志检查你的答案。需要多少内存引用来执行每次查找？

3. 根据你对缓存内存的工作原理的理解，你认为对页表的内存引用如何在缓存中工作？它们是否会导致大量的缓存命中（并导致快速访问）或者很多未命中（并导致访问缓慢）？

第 21 章　超越物理内存：机制

到目前为止，我们一直假定地址空间非常小，能放入物理内存。事实上，我们假设每个正在运行的进程的地址空间都能放入内存。我们将放松这些大的假设，并假设我们需要支持许多同时运行的巨大地址空间。

为了达到这个目的，需要在内存层级（memory hierarchy）上再加一层。到目前为止，我们一直假设所有页都常驻在物理内存中。但是，为了支持更大的地址空间，操作系统需要把当前没有在用的那部分地址空间找个地方存储起来。一般来说，这个地方有一个特点，那就是比内存有更大的容量。因此，一般来说也更慢（如果它足够快，我们就可以像使用内存一样使用，对吗？）。在现代系统中，硬盘（hard disk drive）通常能够满足这个需求。因此，在我们的存储层级结构中，大而慢的硬盘位于底层，内存之上。那么我们的关键问题是：

关键问题：如何超越物理内存

操作系统如何利用大而慢的设备，透明地提供巨大虚拟地址空间的假象？

你可能会问一个问题：为什么我们要为进程支持巨大的地址空间？答案还是方便和易用性。有了巨大的地址空间，你不必担心程序的数据结构是否有足够空间存储，只需自然地编写程序，根据需要分配内存。这是操作系统提供的一个强大的假象，使你的生活简单很多。别客气！一个反面例子是，一些早期系统使用“内存覆盖（memory overlays）”，它需要程序员根据需要手动移入或移出内存中的代码或数据[D97]。设想这样的场景：在调用函数或访问某些数据之前，你需要先安排将代码或数据移入内存。

补充：存储技术

稍后将深入介绍 I/O 设备如何运行。所以少安毋躁！当然，这个较慢的设备可以是硬盘，也可以是一些更新的设备，比如基于闪存的 SSD。我们也会讨论这些内容。但是现在，只要假设有一个大而较慢的设备，可以利用它来构建巨大虚拟内存的假象，甚至比物理内存本身更大。

不仅是一个进程，增加交换空间让操作系统为多个并发运行的进程都提供巨大地址空间的假象。多道程序（能够“同时”运行多个程序，更好地利用机器资源）的出现，强烈要求能够换出一些页，因为早期的机器显然不能将所有进程需要的所有页同时放在内存中。因此，多道程序和易用性都需要操作系统支持比物理内存更大的地址空间。这是所有现代虚拟内存系统都会做的事情，也是现在我们要进一步学习的内容。

21.1　交换空间

我们要做的第一件事情就是，在硬盘上开辟一部分空间用于物理页的移入和移出。在

操作系统中，一般这样的空间称为交换空间（swap space），因为我们将内存中的页交换到其中，并在需要的时候又交换回去。因此，我们会假设操作系统能够以页大小为单元读取或者写入交换空间。为了达到这个目的，操作系统需要记住给定页的硬盘地址（disk address）。

交换空间的大小是非常重要的，它决定了系统在某一时刻能够使用的最大内存页数。简单起见，现在假设它非常大。

在小例子中（见图 21.1），你可以看到一个 4 页的物理内存和一个 8 页的交换空间。在这个例子中，3 个进程（进程 0、进程 1 和进程 2）主动共享物理内存。但 3 个中的每一个，都只有一部分有效页在内存中，剩下的在硬盘的交换空间中。第 4 个进程（进程 3）的所有页都被交换到硬盘上，因此很清楚它目前没有运行。有一块交换空间是空闲的。即使通过这个小例子，你应该也能看出，使用交换空间如何让系统假装内存比实际物理内存更大。

我们需要注意，交换空间不是唯一的硬盘交换目的地。例如，假设运行一个二进制程序（如 ls，或者你自己编译的 main 程序）。这个二进制程序的代码页最开始是在硬盘上，但程序运行的时候，它们被加载到内存中（要么在程序开始运行时全部加载，要么在现代操作系统中，按需要一页一页加载）。但是，如果系统需要在物理内存中腾出空间以满足其他需求，则可以安全地重新使用这些代码页的内存空间，因为稍后它又可以重新从硬盘上的二进制文件加载。

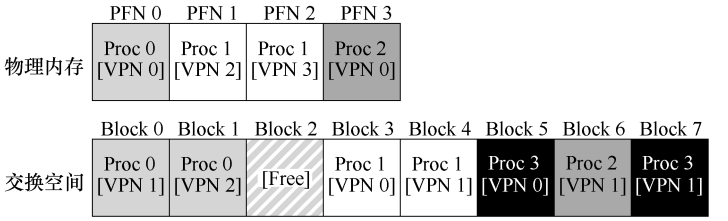

图 21.1 物理内存和交换空间

21.2 存在位

现在我们在硬盘上有一些空间，需要在系统中增加一些更高级的机制，来支持从硬盘交换页。简单起见，假设有一个硬件管理 TLB 的系统。

先回想一下内存引用发生了什么。正在运行的进程生成虚拟内存引用（用于获取指令或访问数据），在这种情况下，硬件将其转换为物理地址，再从内存中获取所需数据。

硬件首先从虚拟地址获得 VPN，检查 TLB 是否匹配（TLB 命中），如果命中，则获得最终的物理地址并从内存中取回。这希望是常见情形，因为它很快（不需要额外的内存访问）。

如果在 TLB 中找不到 VPN（即 TLB 未命中），则硬件在内存中查找页表（使用页表基址寄存器），并使用 VPN 查找该页的页表项（PTE）作为索引。如果页有效且存在于物理内存中，则硬件从 PTE 中获得 PFN，将其插入 TLB，并重试该指令，这次产生 TLB 命中。到现在为止还挺好。

但是，如果希望允许页交换到硬盘，必须添加更多的机制。具体来说，当硬件在 PTE 中查找时，可能发现页不在物理内存中。硬件（或操作系统，在软件管理 TLB 时）判断是

否在内存中的方法，是通过页表项中的一条新信息，即存在位（present bit）。如果存在位设置为 1，则表示该页存在于物理内存中，并且所有内容都如上所述进行。如果存在位设置为零，则页不在内存中，而在硬盘上。访问不在物理内存中的页，这种行为通常被称为页错误（page fault）。

补充：交换术语及其他

对于不同的机器和操作系统，虚拟内存系统的术语可能会有点令人困惑和不同。例如，页错误（page fault）一般是指对页表引用时产生某种错误：这可能包括在这里讨论的错误类型，即页不存在的错误，但有时指的是内存非法访问。事实上，我们将这种完全合法的访问（页被映射到进程的虚拟地址空间，但此时不在物理内存中）称为“错误”是很奇怪的。实际上，它应该被称为“页未命中（page miss）”。但是通常，当人们说一个程序“页错误”时，意味着它正在访问的虚拟地址空间的一部分，被操作系统交换到了硬盘上。

我们怀疑这种行为之所以被称为“错误”，是因为操作系统中的处理机制。当一些不寻常的事情发生的时候，即硬件不知道如何处理的时候，硬件只是简单地把控制权交给操作系统，希望操作系统能够解决。在这种情况下，进程想要访问的页不在内存中。硬件唯一能做的就是触发异常，操作系统从开始接管。由于这与进程执行非法操作处理流程一样，所以我们把这个活动称为“错误”，这也许并不奇怪。

在页错误时，操作系统被唤起来处理页错误。一段称为“页错误处理程序（page-fault handler）”的代码会执行，来处理页错误，接下来就会讲。

21.3 页错误

回想一下，在 TLB 未命中的情况下，我们有两种类型的系统：硬件管理的 TLB（硬件在页表中找到需要的转换映射）和软件管理的 TLB（操作系统执行查找过程）。不论在哪种系统中，如果页不存在，都由操作系统负责处理页错误。操作系统的页错误处理程序（page-fault handler）确定要做什么。几乎所有的系统都在软件中处理页错误。即使是硬件管理的 TLB，硬件也信任操作系统来管理这个重要的任务。

如果一个页不存在，它已被交换到硬盘，在处理页错误的时候，操作系统需要将该页交换到内存中。那么，问题来了：操作系统如何知道所需的页在哪儿？在许多系统中，页表是存储这些信息最自然的地方。因此，操作系统可以用 PTE 中的某些位来存储硬盘地址，这些位通常用来存储像页的 PFN 这样的数据。当操作系统接收到页错误时，它会在 PTE 中查找地址，并将请求发送到硬盘，将页读取到内存中。

补充：为什么硬件不能处理页错误

我们从 TLB 的经验中得知，硬件设计者不愿意信任操作系统做所有事情。那么为什么他们相信操作系统来处理页错误呢？有几个主要原因。首先，页错误导致的硬盘操作很慢。即使操作系统需要很长时间来处理故障，执行大量的指令，但相比于硬盘操作，这些额外开销是很小的。其次，为了能够处理页故障，硬件必须了解交换空间，如何向硬盘发起 I/O 操作，以及很多它当前所不知道的细节。因此，由于性能和简单的原因，操作系统来处理页错误，即使硬件人员也很开心。

当硬盘 I/O 完成时，操作系统会更新页表，将此页标记为存在，更新页表项（PTE）的 PFN 字段以记录新获取页的内存位置，并重试指令。下一次重新访问 TLB 还是未命中，然而这次因为页在内存中，因此会将页表中的地址更新到 TLB 中（也可以在处理页错误时更新 TLB 以避免此步骤）。最后的重试操作会在 TLB 中找到转换映射，从已转换的内存物理地址，获取所需的数据或指令。

请注意，当 I/O 在运行时，进程将处于阻塞（blocked）状态。因此，当页错误正常处理时，操作系统可以自由地运行其他可执行的进程。因为 I/O 操作是昂贵的，一个进程进行 I/O（页错误）时会执行另一个进程，这种交叠（overlap）是多道程序系统充分利用硬件的一种方式。

21.4 内存满了怎么办

在上面描述的过程中，你可能会注意到，我们假设有足够的空闲内存来从存储交换空间换入（page in）的页。当然，情况可能并非如此。内存可能已满（或接近满了）。因此，操作系统可能希望先交换出（page out）一个或多个页，以便为操作系统即将交换入的新页留出空间。选择哪些页被交换出或被替换（replace）的过程，被称为页交换策略（page-replacement policy）。

事实表明，人们在创建好页交换策略上投入了许多思考，因为换出不合适的页会导致程序性能上的巨大损失，也会导致程序以类似硬盘的速度运行而不是以类似内存的速度。在现有的技术条件下，这意味着程序可能会运行慢 10000～100000 倍。因此，这样的策略是我们应该详细研究的。实际上，这也正是我们下一章要做的。现在，我们只要知道有这样的策略存在，建立在之前描述的机制之上。

21.5 页错误处理流程

有了这些知识，我们现在就可以粗略地描绘内存访问的完整流程。换言之，如果有人问你：“当程序从内存中读取数据会发生什么？”，你应该对所有不同的可能性有了很好的概念。有关详细信息，请参见图 21.2 和图 21.3 中的控制流。图 21.2 展示了硬件在地址转换过程中所做的工作，图 21.3 展示了操作系统在页错误时所做的工作。

```
1    VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2    (Success, TlbEntry) = TLB_Lookup(VPN)
3    if (Success == True)     // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True)
5            Offset     = VirtualAddress & OFFSET_MASK
6            PhysAddr   = (TlbEntry.PFN << SHIFT) | Offset
7            Register   = AccessMemory(PhysAddr)
8        else
9            RaiseException(PROTECTION_FAULT)
```

```
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE  =  AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGE_FAULT)
```

图 21.2 页错误控制流算法（硬件）

从图 21.2 的硬件控制流图中，可以注意到当 TLB 未命中发生的时候有 3 种重要情景。第一种情况，该页存在（present）且有效（valid）（第 18～21 行）。在这种情况下，TLB 未命中处理程序可以简单地从 PTE 中获取 PFN，然后重试指令（这次 TLB 会命中），并因此继续前面描述的流程。第二种情况（第 22～23 行），页错误处理程序需要运行。虽然这是进程可以访问的合法页（毕竟是有效的），但它并不在物理内存中。第三种情况，访问的是一个无效页，可能由于程序中的错误（第 13～14 行）。在这种情况下，PTE 中的其他位都不重要了。硬件捕获这个非法访问，操作系统陷阱处理程序运行，可能会杀死非法进程。

从图 21.3 的软件控制流中，可以看到为了处理页错误，操作系统大致做了什么。首先，操作系统必须为将要换入的页找到一个物理帧，如果没有这样的物理帧，我们将不得不等待交换算法运行，并从内存中踢出一些页，释放帧供这里使用。在获得物理帧后，处理程序发出 I/O 请求从交换空间读取页。最后，当这个慢操作完成时，操作系统更新页表并重试指令。重试将导致 TLB 未命中，然后再一次重试时，TLB 命中，此时硬件将能够访问所需的值。

```
1   PFN = FindFreePhysicalPage()
2   if (PFN == -1)              // no free page found
3       PFN = EvictPage()       // run replacement algorithm
4   DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5   PTE.present = True          // update page table with present
6   PTE.PFN     = PFN           // bit and translation (PFN)
7   RetryInstruction()          // retry instruction
```

图 21.3 页错误控制流算法（软件）

21.6 交换何时真正发生

到目前为止，我们一直描述的是操作系统会等到内存已经完全满了以后才会执行交换流程，然后才替换（踢出）一个页为其他页腾出空间。正如你想象的那样，这有点不切实

际的，因为操作系统可以更主动地预留一小部分空闲内存。

为了保证有少量的空闲内存，大多数操作系统会设置高水位线（High Watermark，HW）和低水位线（Low Watermark，LW），来帮助决定何时从内存中清除页。原理是这样：当操作系统发现有少于 LW 个页可用时，后台负责释放内存的线程会开始运行，直到有 HW 个可用的物理页。这个后台线程有时称为交换守护进程（swap daemon）或页守护进程（page daemon）[①]，它然后会很开心地进入休眠状态，因为它毕竟为操作系统释放了一些内存。

通过同时执行多个交换过程，我们可以进行一些性能优化。例如，许多系统会把多个要写入的页聚集（cluster）或分组（group），同时写入到交换区间，从而提高硬盘的效率[LL82]。我们稍后在讨论硬盘时将会看到，这种合并操作减少了硬盘的寻道和旋转开销，从而显著提高了性能。

为了配合后台的分页线程，图 21.3 中的控制流需要稍作修改。交换算法需要先简单检查是否有空闲页，而不是直接执行替换。如果没有空闲页，会通知后台分页线程按需要释放页。当线程释放一定数目的页时，它会重新唤醒原来的线程，然后就可以把需要的页交换进内存，继续它的工作。

提示：把一些工作放在后台

当你有一些工作要做的时候，把这些工作放在后台（background）运行是一个好注意，可以提高效率，并允许将这些操作合并执行。操作系统通常在后台执行很多工作。例如，在将数据写入硬盘之前，许多系统在内存中缓冲要写入的数据。这样做有很多好处：提高硬盘效率，因为硬盘现在可以一次写入多次要写入的数据，因此能够更好地调度这些写入。优化了写入延迟，因为数据写入到内存就可以返回。可能减少某些操作，因为写入操作可能不需要写入硬盘（例如，如果文件马上又被删除），也能更好地利用系统空闲时间（idle time），因为系统可以在空闲时完成后台工作，从而更好地利用硬件资源[G+95]。

21.7 小结

在这个简短的一章中，我们介绍了访问超出物理内存大小时的一些概念。要做到这一点，在页表结构中需要添加额外信息，比如增加一个存在位（present bit，或者其他类似机制），告诉我们页是不是在内存中。如果不存在，则操作系统页错误处理程序（page-fault handler）会运行以处理页错误（page fault），从而将需要的页从硬盘读取到内存，可能还需要先换出内存中的一些页，为即将换入的页腾出空间。

回想一下，很重要的是（并且令人惊讶的是），这些行为对进程都是透明的。对进程而言，它只是访问自己私有的、连续的虚拟内存。在后台，物理页被放置在物理内存中的任意（非连续）位置，有时它们甚至不在内存中，需要从硬盘取回。虽然我们希望在一般情况下内存访问速度很快，但在某些情况下，它需要多个硬盘操作的时间。像执行单条指令这样简单的事情，在最坏的情况下，可能需要很多毫秒才能完成。

① “守护进程（daemon）”这个词通常发音为“demon”，它是一个古老的术语，用于后台线程或过程，它可以做一些有用的事情。事实表明，该术语的来源是 Multics [CS94]。

参考资料

[CS94]“Take Our Word For It”
F. Corbato and R. Steinberg
Richard Steinberg 写道：“有人问我守护进程（daemon）这个词什么时候开始用于计算。根据我的研究，最好的结果是，这个词在 1963 年被你的团队在使用 IBM 7094 的 Project MAC 中首次使用。”Corbato 教授回答说：“我们使用守护进程这个词的灵感来源于物理学和热力学的麦克斯韦尔守护进程（Maxwell’s daemon，我的背景是物理学）。麦克斯韦尔守护进程是一个虚构的代理，帮助分拣不同速度的分子，并在后台不知疲倦地工作。我们别出心裁地开始使用守护进程来描述后台进程，这些进程不知疲倦地执行系统任务。”

[D97]“Before Memory Was Virtual”Peter Denning
From In the Beginning: Recollections of Software Pioneers, Wiley, November 1997
优秀的历史性作品，作者是虚拟内存和工作团队的先驱者之一。

[G+95]“Idleness is not sloth”
Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, John Wilkes USENIX ATC ’95, New Orleans, Louisiana
有趣且易于阅读的讨论，关于如何在系统中更好地利用空闲时间，有很多很好的例子。

[LL82]“Virtual Memory Management in the VAX/VMS Operating System”Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
这不是第一个使用这种聚集机制的地方，却是对这种机制如何工作的清晰而简单的解释。

第 22 章　超越物理内存：策略

在虚拟内存管理程序中，如果拥有大量空闲内存，操作就会变得很容易。页错误发生了，你在空闲页列表中找到空闲页，将它分配给不在内存中的页。嘿，操作系统，恭喜！ 你又成功了。

遗憾的是，当内存不够时事情会变得更有趣。在这种情况下，由于内存压力（memory pressure）迫使操作系统换出（paging out）一些页，为常用的页腾出空间。确定要踢出（evict）哪个页（或哪些页）封装在操作系统的替换策略（replacement policy）中。历史上，这是早期的虚拟内存系统要做的最重要的决定之一，因为旧系统的物理内存非常小。至少，有一些策略是非常值得了解的。因此我们的问题如下所示。

关键问题：如何决定踢出哪个页

操作系统如何决定从内存中踢出哪一页（或哪几页）？这个决定由系统的替换策略做出，替换策略通常会遵循一些通用的原则（下面将会讨论），但也会包括一些调整，以避免特殊情况下的行为。

22.1　缓存管理

在深入研究策略之前，先详细描述一下我们要解决的问题。由于内存只包含系统中所有页的子集，因此可以将其视为系统中虚拟内存页的缓存（cache）。因此，在为这个缓存选择替换策略时，我们的目标是让缓存未命中（cache miss）最少，即使得从磁盘获取页的次数最少。或者，可以将目标看成让缓存命中（cache hit）最多，即在内存中找到待访问页的次数最多。

知道了缓存命中和未命中的次数，就可以计算程序的平均内存访问时间（Average Memory Access Time，AMAT，计算机架构师衡量硬件缓存的指标 [HP06]）。具体来说，给定这些值，可以按照如下公式计算 *AMAT*：

$$AMAT = (P_{\text{Hit}} \cdot T_{\text{M}}) + (P_{\text{Miss}} \cdot T_{\text{D}})$$

其中 T_{M} 表示访问内存的成本，T_{D} 表示访问磁盘的成本，P_{Hit} 表示在缓存中找到数据的概率（命中），P_{Miss} 表示在缓存中找不到数据的概率（未命中）。P_{Hit} 和 P_{Miss} 从 0.0 变化到 1.0，并且 $P_{\text{Miss}} + P_{\text{Hit}} = 1.0$。

例如，假设有一个机器有小型地址空间：4KB，每页 256 字节。因此，虚拟地址由两部分组成：一个 4 位 VPN（最高有效位）和一个 8 位偏移量（最低有效位）。因此，本例中的一个进程可以访问总共 $2^4=16$ 个虚拟页。在这个例子中，该进程将产生以下内存引用（即虚拟地址）0x000，0x100，0x200，0x300，0x400，0x500，0x600，0x700，0x800，0x900。

这些虚拟地址指向地址空间中前 10 页的每一页的第一个字节（页号是每个虚拟地址的第一个十六进制数字）。

让我们进一步假设，除了虚拟页 3 之外，所有页都已经在内存中。因此，我们的内存引用序列将遇到以下行为：命中，命中，命中，未命中，命中，命中，命中，命中，命中。我们可以计算命中率（hit rate，在内存中找到引用的百分比）：90%（P_{Hit} = 0.9），因为 10 个引用中有 9 个在内存中。未命中率（miss rate）显然是 10%（P_{Miss} = 0.1）。

要计算 AMAT，需要知道访问内存的成本和访问磁盘的成本。假设访问内存（TM）的成本约为 100ns，并且访问磁盘（TD）的成本大约为 10ms，则我们有以下 AMAT：0.9×100ns + 0.1×10ms，即 90ns + 1ms 或 1.0009ms，或约 1ms。如果我们的命中率是 99.9%（P_{Miss} = 0.001），结果是完全不同的：AMAT 是 10.1μs，大约快 100 倍。当命中率接近 100%时，AMAT 接近 100ns。

遗憾的是，正如你在这个例子中看到的，在现代系统中，磁盘访问的成本非常高，即使很小概率的未命中也会拉低正在运行的程序的总体 AMAT。显然，我们必须尽可能地避免缓存未命中，避免程序以磁盘的速度运行。要做到这一点，有一种方法就是仔细开发一个聪明的策略，像我们现在所做的一样。

22.2 最优替换策略

为了更好地理解一个特定的替换策略是如何工作的，将它与最好的替换策略进行比较是很好的方法。事实证明，这样一个最优（optimal）策略是 Belady 多年前开发的[B66]（原来这个策略叫作 MIN）。最优替换策略能达到总体未命中数量最少。Belady 展示了一个简单的方法（但遗憾的是，很难实现！），即替换内存中在最远将来才会被访问到的页，可以达到缓存未命中率最低。

提示：与最优策略对比非常有用

虽然最优策略非常不切实际，但作为仿真或其他研究的比较者还是非常有用的。比如，单说你喜欢的新算法有 80%的命中率是没有意义的，但加上最优算法只有 82%的命中率（因此你的新方法非常接近最优），就会使得结果很有意义，并给出了它的上下文。因此，在你进行的任何研究中，知道最优策略可以方便进行对比，知道你的策略有多大的改进空间，也用于决定当策略已经非常接近最优策略时，停止做无谓的优化[AD03]。

希望最优策略背后的想法你能理解。这样想：如果你不得不踢出一些页，为什么不踢出在最远将来才会访问的页呢？这样做基本上是说，缓存中所有其他页都比这个页重要。道理很简单：在引用最远将来会访问的页之前，你肯定会引用其他页。

我们追踪一个简单的例子，来理解最优策略的决定。假设一个程序按照以下顺序访问虚拟页：0，1，2，0，1，3，0，3，1，2，1。表 22.1 展示了最优的策略，这里假设缓存可以存 3 个页。

在表 22.1 中，可以看到以下操作。不要惊讶，前 3 个访问是未命中，因为缓存开始是

空的。这种未命中有时也称作冷启动未命中（cold-start miss，或强制未命中，compulsory miss）。然后我们再次引用页 0 和 1，它们都在缓存中。最后，我们又有一个缓存未命中（页 3），但这时缓存已满，必须进行替换！这引出了一个问题：我们应该替换哪个页？使用最优策略，我们检查当前缓存中每个页（0、1 和 2）未来访问情况，可以看到页 0 马上被访问，页 1 稍后被访问，页 2 在最远的将来被访问。因此，最优策略的选择很简单：踢出页面 2，结果是缓存中的页面是 0、1 和 3。接下来的 3 个引用是命中的，然后又访问到被我们之前踢出的页 2，那么又有一个未命中。这里，最优策略再次检查缓存页（0、1 和 3）中每个页面的未来被访问情况，并且看到只要不踢出页 1（即将被访问）就可以。这个例子显示了页 3 被踢出，虽然踢出 0 也是可以的。最后，我们命中页 1，追踪完成。

表 22.1 追踪最优策略

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

补充：缓存未命中的类型

在计算机体系结构世界中，架构师有时会将未命中分为 3 类：强制性、容量和冲突未命中，有时称为 3C [H87]。发生强制性（compulsory miss）未命中（或冷启动未命中，cold-start miss [EF78]）是因为缓存开始是空的，而这是对项目的第一次引用。与此不同，由于缓存的空间不足而不得不踢出一个项目以将新项目引入缓存，就发生了容量未命中（capacity miss）。第三种类型的未命中（冲突未命中，conflict miss）出现在硬件中，因为硬件缓存中对项的放置位置有限制，这是由于所谓的集合关联性（set-associativity）。它不会出现在操作系统页面缓存中，因为这样的缓存总是完全关联的（fully-associative），即对页面可以放置的内存位置没有限制。详情请见 H&P [HP06]。

我们同时计算缓存命中率：有 6 次命中和 5 次未命中，那么缓存命中率 $\frac{Hits}{Hits + Misses}$ 是 $\frac{6}{6+5}$，或 54.5%。也可以计算命中率中除去强制未命中（即忽略页的第一次未命中），那么命中率为 81.8%。

遗憾的是，正如我们之前在开发调度策略时所看到的那样，未来的访问是无法知道的，

你无法为通用操作系统实现最优策略[①]。因此，在开发一个真正的、可实现的策略时，我们将聚焦于寻找其他决定把哪个页面踢出的方法。因此，最优策略只能作为比较，知道我们的策略有多接近“完美”。

22.3 简单策略：FIFO

许多早期的系统避免了尝试达到最优的复杂性，采用了非常简单的替换策略。例如，一些系统使用 FIFO（先入先出）替换策略。页在进入系统时，简单地放入一个队列。当发生替换时，队列尾部的页（“先入”页）被踢出。FIFO 有一个很大的优势：实现相当简单。

让我们来看看 FIFO 策略如何执行这过程（见表 22.2）。我们再次开始追踪 3 个页面 0、1 和 2。首先是强制性未命中，然后命中页 0 和 1。接下来，引用页 3，缓存未命中。使用 FIFO 策略决定替换哪个页面是很容易的：选择第一个进入的页，这里是页 0（表中的缓存状态列是按照先进先出顺序，最左侧是第一个进来的页），遗憾的是，我们的下一个访问还是页 0，导致另一次未命中和替换（替换页 1）。然后我们命中页 3，但是未命中页 1 和 2，最后命中页 3。

表 22.2 追踪 FIFO 策略

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

对比 FIFO 和最优策略，FIFO 明显不如最优策略，FIFO 命中率只有 36.4%（不包括强制性未命中为 57.1%）。先进先出（FIFO）根本无法确定页的重要性：即使页 0 已被多次访问，FIFO 仍然会将其踢出，因为它是第一个进入内存的。

补充：Belady 的异常

Belady（最优策略发明者）及其同事发现了一个有意思的引用序列[BNS69]。内存引用顺序是：1，2，3，4，1，2，5，1，2，3，4，5。他们正在研究的替换策略是 FIFO。有趣的问题：当缓存大小从 3 变成 4 时，缓存命中率如何变化？

① 如果你可以，请告诉我们，我们可以一起发财，或者，像 “发现” 冷聚变的科学家一样，被众人所讽刺和嘲笑[FP89]。

一般来说，当缓存变大时，缓存命中率是会提高的（变好）。但在这个例子，采用 FIFO，命中率反而下降了！你可以自己计算一下缓存命中和未命中次数。这种奇怪的现象被称为 Belady 的异常（Belady's Anomaly）。

其他一些策略，比如 LRU，不会遇到这个问题。可以猜猜为什么？事实证明，LRU 具有所谓的栈特性（stack property）[M+70]。对于具有这个性质的算法，大小为 $N+1$ 的缓存自然包括大小为 N 的缓存的内容。因此，当增加缓存大小时，缓存命中率至少保证不变，有可能提高。先进先出（FIFO）和随机（Random）等显然没有栈特性，因此容易出现异常行为。

22.4 另一简单策略：随机

另一个类似的替换策略是随机，在内存满的时候它随机选择一个页进行替换。随机具有类似于 FIFO 的属性。实现起来很简单，但是它在挑选替换哪个页时不够智能。让我们来看看随机策略在我们著名的例子上的引用流程（见表 22.3）。

表 22.3 追踪随机策略

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

当然，随机的表现完全取决于多幸运（或不幸）。在上面的例子中，随机比 FIFO 好一点，比最优的差一点。事实上，我们可以运行数千次的随机实验，求得一个平均的结果。图 22.1 显示了 10000 次试验后随机策略的平均命中率，每次试验都有不同的随机种子。正如你所看到的，有些时候（仅仅 40%的概率），随机和最优策略一样好，在上述例子中，命中内存

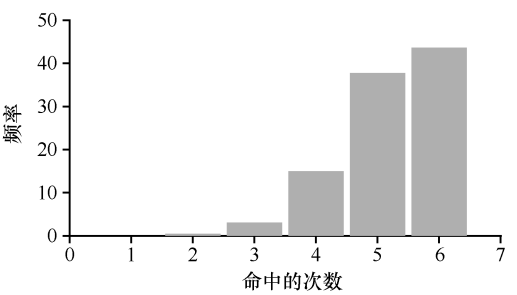

图 22.1 随机策略在 10000 次尝试下的表现

的次数是 6 次。有时候情况会更糟糕，只有 2 次或更少。随机策略取决于当时的运气。

22.5 利用历史数据：LRU

遗憾的是，任何像 FIFO 或随机这样简单的策略都可能会有一个共同的问题：它可能会踢出一个重要的页，而这个页马上要被引用。先进先出（FIFO）将先进入的页踢出。如果这恰好是一个包含重要代码或数据结构的页，它还是会被踢出，尽管它很快会被重新载入。因此，FIFO、Random 和类似的策略不太可能达到最优，需要更智能的策略。

正如在调度策略所做的那样，为了提高后续的命中率，我们再次通过历史的访问情况作为参考。例如，如果某个程序在过去访问过某个页，则很有可能在不久的将来会再次访问该页。

页替换策略可以使用的一个历史信息是频率（frequency）。如果一个页被访问了很多次，也许它不应该被替换，因为它显然更有价值。页更常用的属性是访问的近期性（recency），越近被访问过的页，也许再次访问的可能性也就越大。

这一系列的策略是基于人们所说的局部性原则（principle of locality）[D70]，基本上只是对程序及其行为的观察。这个原理简单地说就是程序倾向于频繁地访问某些代码（例如循环）和数据结构（例如循环访问的数组）。因此，我们应该尝试用历史数据来确定哪些页面更重要，并在需要踢出页时将这些页保存在内存中。

因此，一系列简单的基于历史的算法诞生了。“最不经常使用”（Least-Frequently-Used，LFU）策略会替换最不经常使用的页。同样，“最少最近使用”（Least-Recently-Used，LRU）策略替换最近最少使用的页面。这些算法很容易记住：一旦知道这个名字，就能确切知道它是什么，这种名字就非常好。

补充：局部性类型

程序倾向于表现出两种类型的局部。第一种是空间局部性（spatial locality），它指出如果页 P 被访问，可能围绕它的页（比如 P−1 或 P + 1）也会被访问。第二种是时间局部性（temporal locality），它指出近期访问过的页面很可能在不久的将来再次访问。假设存在这些类型的局部性，对硬件系统的缓存层次结构起着重要作用，硬件系统部署了许多级别的指令、数据和地址转换缓存，以便在存在此类局部性时，能帮助程序快速运行。

当然，通常所说的局部性原则（principle of locality）并不是硬性规定，所有的程序都必须遵守。事实上，一些程序以相当随机的方式访问内存（或磁盘），并且在其访问序列中不显示太多或完全没有局部性。因此，尽管在设计任何类型的缓存（硬件或软件）时，局部性都是一件好事，但它并不能保证成功。相反，它是一种经常证明在计算机系统设计中有用的启发式方法。

为了更好地理解 LRU，我们来看看 LRU 如何在示例引用序列上执行。表 22.4 展示了结果。从表中，可以看到 LRU 如何利用历史记录，比无状态策略（如随机或 FIFO）做得更好。在这个例子中，当第一次需要替换页时，LRU 会踢出页 2，因为 0 和 1 的访问时间更近。然后它替换页 0，因为 1 和 3 最近被访问过。在这两种情况下，基于历史的 LRU 的决