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。

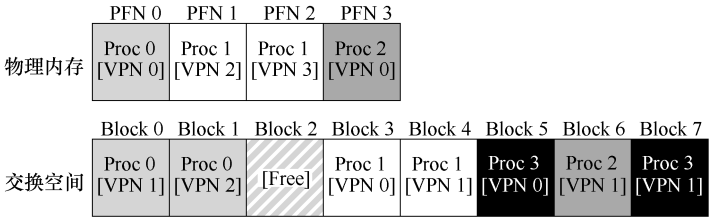

图 21.1 物理内存和交换空间

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。