5.2 页式存储管理

1. 分页基本方法

分页 (paging) 内存管理方案允许进程的物理地址空间可以是非连续的。分页避免了将不同大小的内存块匹配到交换空间上这样的麻烦。当位于内存中的代码或数据需要换出时，必须先在备份区上找到空间，这时间题就产生了。备份区也有前面所述与内存相关的碎片问题，只不过访问更慢，因此不适宜采用合并。所以各种形式的分页由于其优越性，通常为绝大多数操作系统都采用。传统上，分页支持一直是由硬件来处理的。然而，最近的设计是通过将硬件和操作系统相配合来实现分页(尤其是在 64 位微处理器上)。

实现分页的基本方法涉及将进程逻辑地址空间分成大小相等的区，每个分区称为页(page)，页从 0 开始依次编号。同时把主存物理地址空间也分成大小相等的区，每个区是一个帧(frame，也叫物理块或页框)。帧大小与页大小相等。运行一个有 n 页大小的程序，需要找到 n 个空闲帧来存放程序。

与此对应分页存储器的逻辑地址由两部分组成：页号和页偏移。逻辑地址是连续的，用户在编制程序时仍使用相对地址，不必考虑如何分页，由硬件地址转换机制和操作系统的管理需要来决定页尺寸，从而确定主存的分块大小。进程在主存中的每个帧内的地址是连续的，但帧之间的地址可以不连续。

页的大小通常为 2 的幂次，根据计算机结构的不同，其每页大小从 512B ~ 16MB 不等。在 Linux x86 中一个帧和一个页的大小为 4K。选择页的大小为 2 的幂可以方便地将逻辑地址转换为页号和页偏移。如果逻辑地址空间为 2^m，且页大小为 2^n 单元(字节或字)，那么逻辑地址的高 m-n 位表示页号，而低 n 位表示页偏移。

页表是操作系统为进程建立的，是进程的逻辑页和主存对应物理帧的对照表。页表包含每页所在物理内存的基地址，其每一栏指明进程中的一个页和分得的帧之间的对应关系。通过页表可以把逻辑地址转化成物理地址，送交物理单元。内存的分页模型如图 5.11 所示。

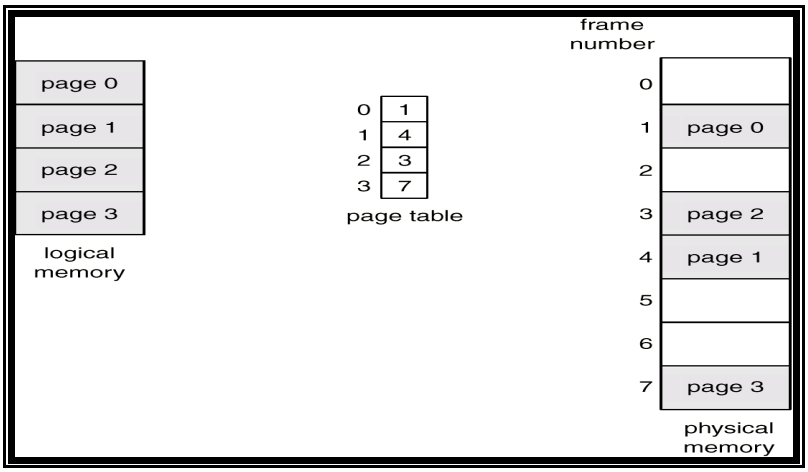

图 5.11 逻辑内存和物理内存的分页模型

值得注意的是，分页其实也是一种动态重定位。每个逻辑地址绑定为一定的物理地址。采用分页类似于使用一组重定位寄存器，每个基地址对应着一个内存帧。

所以分页的重要特点是用户视角的内存和实际的物理内存的分离。用户程序将内存作为一整块来处理，而且它只包括这一个进程。事实上，一个用户程序与其他程序一起，分布在物理内存上。用户视角的内存和实际的物理内存的差异是通过地址转换协调的。逻辑地址转变成物理地址。这种映射是用户所不知道的，但是受操作系统所控制。注意用户进程根据定

义是不能访问非它所占用的内存的。它无法访问其页表所规定之外的内存，页表只包括进程所拥有的那些页。

由于操作系统管理物理内存，它必须知道物理内存的分配细节：哪些帧已占用，哪些帧可用，总共有多少帧，等等。这些信息通常保存在称为帧表的数据结构中。在帧表(frame table) 中，每个条目对应着一个帧，以表示该帧是空闲还是已占用，如果占用，是被哪个(或哪些)进程的哪个页所占用。

另外，操作系统必须意识到用户进程是在用户空间内执行，且所有逻辑地址必须映射到物理地址。如果用户执行一个系统调用(例如进行 I/O)，并提供地址作为参数(如一个缓冲)，那么这个地址必须映射成物理地址。操作系统为每个进程维护一个页表的副本，就如同它需要维护指令计数器和寄存器的内容一样。当操作系统必须手工将逻辑地址映射成物理地址时，这个副本可用来将逻辑地址转换为物理地址。当一个进程可分配到 CPU，CPU 调度程序可以根据该副本来定义页表。因此，分页增加了切换时间。

另外，采用分页技术的一个好处是不会产生外部碎片：每个帧都可以分配给需要它的进程。不过，分页有内部碎片。注意分配是以帧为单元进行的。如果进程所要求的内存并不是页的整数倍，那么最后一个帧就可能用不完。例如，如果页的大小为 2048 B，一个大小为 72776 B 的进程需要 35 个页和 1086 B。进程会得到 36 个帧，因此有 2048-1086=962B 的内部碎片。在最坏情况下，一个需 n 页再加 1B 的进程，需要分配 n+1 个帧，这样几乎产生一个整个帧的内部碎片。

如果进程大小与页大小无关，那么可以推测平均每个进程可能有半页的内部碎片。这一结果意味着小一点的页可能好些。不过，由于页表中的每项也有一定的开销，该开销随着页的增大而降低。而且，磁盘 I/ O 操作随着传输量的增大会更为有效。一般来说，随着时间的推移，页的大小也随着进程、数据和内存的不断增大而增大。现在，页大小通常为 4~8 KB，有的系统可能支持更大的页。有的 CPU 内核可能支持多种页大小。例如，Solaris 根据页所存储的数据，可使用 8KB 或 4MB 的大小的页。

2. 地址映射机制

由 CPU 生成的每个地址分为两个部分：页号(p)和页偏移(d)。如果逻辑地址空间为 2^m，且页大小为 2^n 单元(字节或字)，那么页号 p 为逻辑地址的高 m-n 位，而低 n 位表示页偏移 d。页号用来作为页表的索引，可以对应每个页在物理内存中的基址。偏移同基址相结合，用来确定送入内存单元的物理内存地址。页表其每一栏指明进程中的一个页和分得的帧之间的对应关系，从数学角度来看，页表就是一个函数，其变量是页号，函数值为帧号，通过页表中的基地址和逻辑地址的页内偏移两部分的组合就可以把逻辑地址转换为物理地址。其模型如下图 5.12 所示。

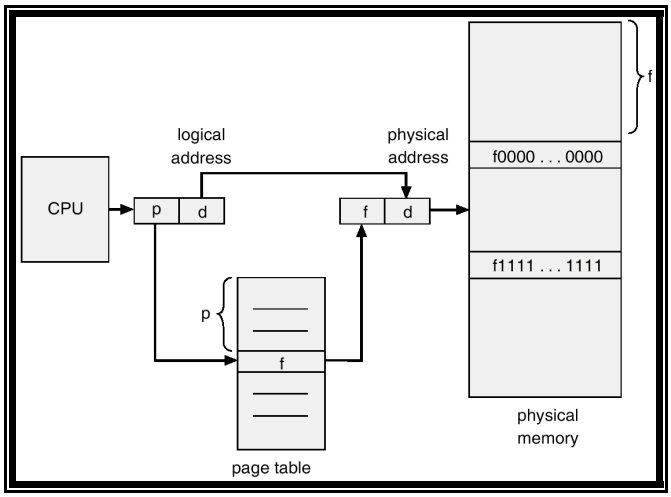

图 5.12 逻辑内存和物理内存的地址映射模型

其中，页表中每个条目所需的位数是由映射后的物理内存空间的大小决定的。根据物理内存大小可以把物理内存地址写成 M 位(能够表示 2^M 的物理地址空间)。在逻辑内存和物理内存的地址映射转换中，它的低 d 位是由逻辑地址的页内偏移得到，剩下的高位的位数则决定了页表中每个条目所需的位数。因为 M=f+d，则高位 f 的位数等于物理内存地址的位数 M 减去逻辑地址的页内偏移位数 d。通常每个页表条目为 4B。一个 4B(32 位)的条目可以指向 2^{32} 个物理帧中的任一个。如果帧为 4KB(2^{12}B)，那么具有 4B 条目的系统可以访问 2^{44}B 大小(或 16TB)的物理内存。

再举一个详细例子来说明映射机制，如图 5.13 所示，如果页大小为 4B，而物理内存为 32B (8 页)，考虑一下用户视角的内存是如何映射到物理内存的。逻辑地址 0 的页号为 0，页偏移为 0。根据页表，可以查到页号 0 对应为帧 5，因此逻辑地址 0 映射为物理地址 20(=(5*4)+0)。逻辑地址 3 (页号为 0，页偏移为 3) 映射为物理地址 23 (= (5*4)+3) 。逻辑地址 4 的页号为 1，页偏移为 0，根据页表，页号 l 对应为帧 6，因此，逻辑地址 4 映射为物理地址 24(= (6*4)+0)。逻辑地址 13 映射为物理地址 9。

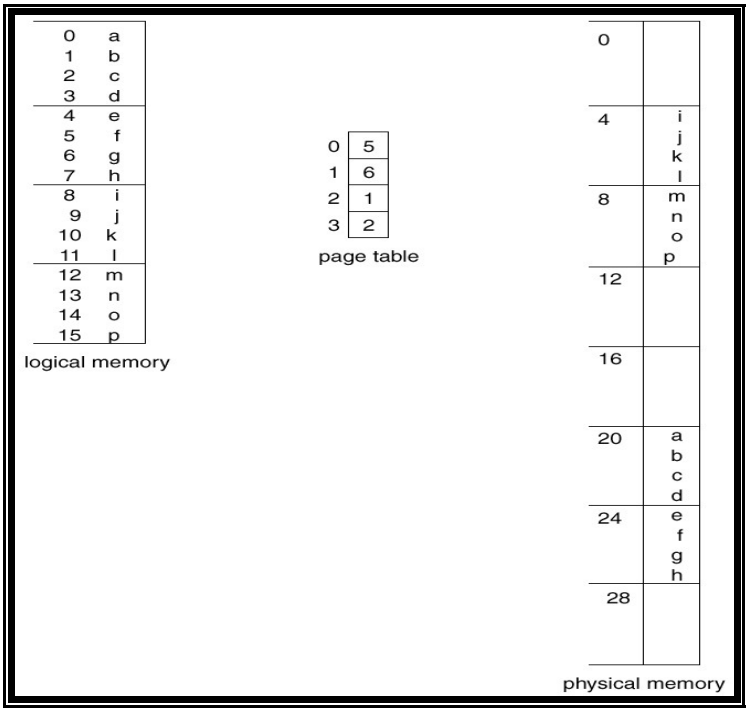

图 5.13 使用 4B 的页对 32B 的内存进行分页的例子

当进程执行时，系统会首先检查该进程的大小(按页来计算)，进程的每页都需要一帧。因此，如果进程需要 n 页，那么内存中至少应有 n 个帧。如果有，那么就可分配给新进程。进程的第一页装入一个已分配的帧，帧号放入进程的页表中。下一页分配给另一帧，其帧号也放入进程的页表中(见图 5.14) 。

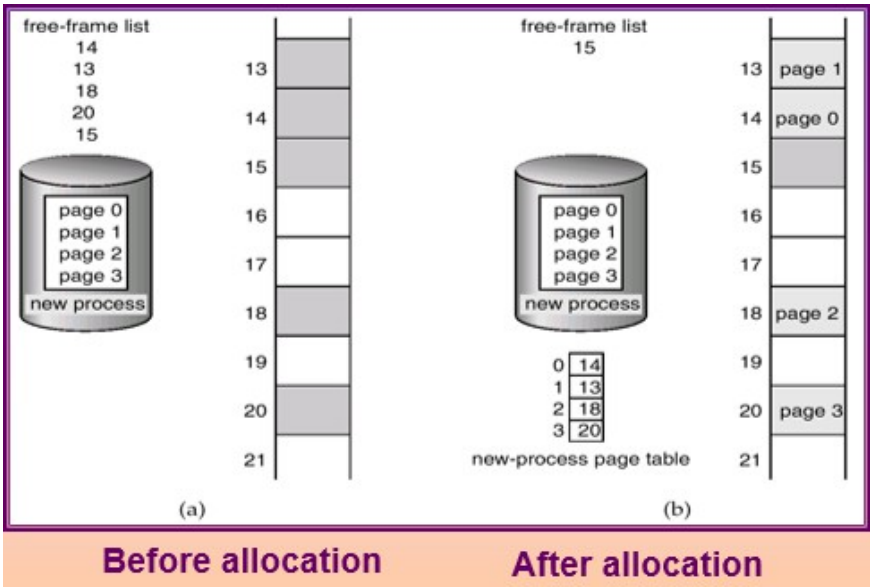

图 5.14 为一个进程分配空间的前后变化

3. 页表的硬件实现

每个操作系统都有自己的方法来保存页表。绝大多数都为每个进程分配一个页表。页表的指针与其他寄存器的值(如指令计数器)一起存入进程控制块中。当调度程序需要启动一个进程时，它必须首先装入用户寄存器，并根据所保存的用户页表来定义正确的硬件页表值。

页表的硬件实现有多种方法。最为简单的一种方法是将页表作为一组专用寄存器(register) 来实现。这些寄存器应用高速逻辑电路来构造，以便有效地进行分页地址的转换。由于对内存的每次访问都要经过分页表，因此效率很重要。CPU 调度程序在装入其他寄存器时，也需要装入这些寄存器。装入或修改页表寄存器的指令是特权级的，因此只有操作系统才可以修改内存映射图。DEC PDP-II 就是这种类型的结构。它的地址有 16 位，而页面大小为 8 KB，因此页表有 8 个条目可放在快速寄存器中。如果页表比较小(例如 256 个条目) ，那么页表使用寄存器还是比较合理的。但是，绝大多数当代计算机都允许页表非常大(如一百万个条目)。对于这些机器，采用快速寄存器来实现页表就不可行了。因而需要将进程页表放在内存中，并设置了专门的硬件页表基寄存器(page-table base register, PTBR)和页表限长寄存器(Page-table length register, PTLR)存放当前运行进程页表的起始地址和页表的长度，以加快地址转换速度。改变页表只需要改变 PTBR 寄存器的指向就可以。

进程运行前由系统把它的页表基地址送入页表基址寄存器，运行时借助于硬件的地址转换，按页面动态地址重定位。当 CPU 获得逻辑地址后，由硬件按设定的页面尺寸分成两部分：页号和页内位移。先从页表基地址寄存器找到页表基地址，再用页号作为索引查找页表，得到对应的页框号。根据物理地址=帧号*块长+页偏移计算出欲访问的主存单元。虽然进程存放在若干不连续的页框中，但在执行过程中总能按正确的物理地址进行存取。

按照给定逻辑地址进行读写操作时，至少访问两次主存：一次访问页面，另一个根据物理地址访问指令或数据。这样，内存访问的速度就减半。在绝大多数情况下，这种延迟是无法忍受的。为了提高速度，对这一问题的解决方案是设置了小但专用且快速的硬件缓冲，这种缓冲称为转换表缓冲区(translation look-aside buffer, TLB)。TLB 是关联的快速内存。

TLB 条目由两部分组成：键(标签)和值。当关联内存根据给定值查找时，它会同时与所有键进行比较。如果找到条目，那么就得到相应的值域。这种查找方式比较快，不过硬件也比较昂贵。通常，TLB 的条目数并不多，通常在 64~1024 之间。

TLB 与页表一起按如下方法使用：TLB 只包括页表中的一小部分条目。当 CPU 产生逻辑地址后，其页号提交给 TLB 。如果找到页号，那么也就得到了帧号，并可用来访问内存。整个任务与不采用内存映射相比，其时间增加不会超过 10%。

如果页码不在 TLB 中(称为 TLB 失效)，那么就需要访问页表。当得到帧号后，就可以用它来访问内存(如图 5.15 所示)。同时，将页号和帧号增加到 TLB 中，这样下次再用时就可很快查找到。如果 TLB 中的条目已满，那么操作系统会选择一个来替换。替换策略有很多，从最近最少使用替换 (LRU) 到随机替换等。另外，有的 TLB 允许有些条目固定下来，也就是说它们不会从 TLB 中被替换。通常内核代码的条目是固定下来的。

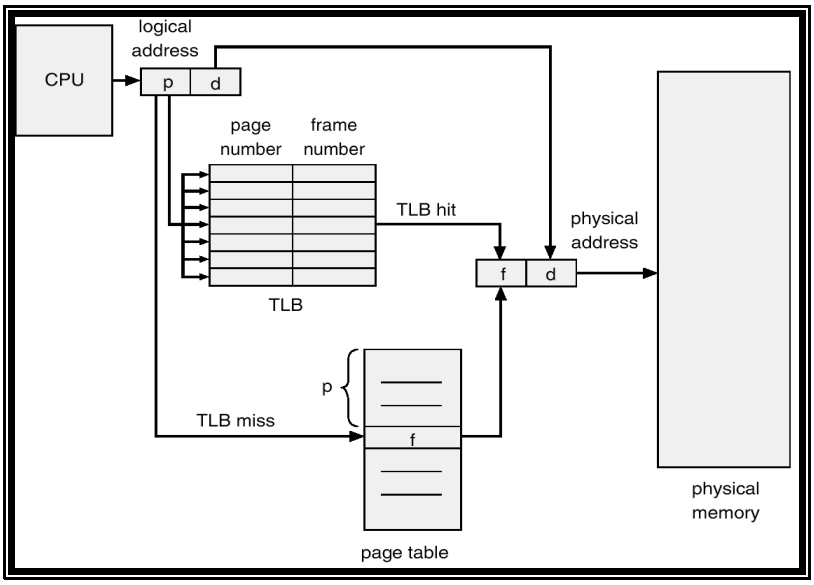

图 5.15 带 TLB 的分页硬件

4. 有效内存访问时间

页号在 TLB 中被查找到的百分比称为命中率ϑ。80%的命中率意味着有 80%的时间，可以在 TLB 中找到所需的页号。假如查找 TLB 需要单位时间，访问内存需要 1 单位时间，如果访问位于 TLB 中的页号，那么采用内存映射访问需要 1+ε。如果不能在 TLB 中找到，那么必须先访问位于内存中的页表以得到帧号(1 单位时间)，并进而访问内存中的所需字节(1 单位时间)，这总共要花费 2+ε 。为了得到有效内存访问时间，必须根据概率来对每种情况进行加权。

有效访问时间 EAT (Effective Access Time) = $(1 + \varepsilon)\ \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$

如果访问一次 TLB 时间为 20ns，访问一次内存时间为 100ns，命中率为 80%，那么

有效内存访问时间= 0.80*120 + 0.20*220 = 140 (ns)

对于这种情况，相当于内存访问速度慢了 40% (从 100~140 ns)。

如果命中率为 98%，那么

有效内存访问时间= 0.98*120 + 0.02*220 = 122 (ns)

由于提高了命中率，现在内存访问速度只慢了 22%。

5. 页表结构

(a) 分级页表

绝大多数现代计算机系统支持大逻辑地址空间(2^{32}~2^{64}) 。在这种情况下，页表本身可以

非常大。例如，设想一下具有 32 位逻辑地址空间的计算机系统。如果系统的页大小为 4KB (2^{12})，那么一个页表可以包含 1 百万个条目($2^{32}/2^{12}$)。假设每个条目有 4B，那么每个进程需要 4MB 物理地址空间来存储页表本身。显然，我们并不可能在内存中连续地分配这个页表。为了减少页表所占用的连续的内存空间，一个简单解决方法是将页表划分为更小部分。划分有许多方法。

一种方法是使用两级分页算法，就是将页表再分页(见图 5.16)，在 Linux 和 Windows x86 中，都是采用了这种机制。

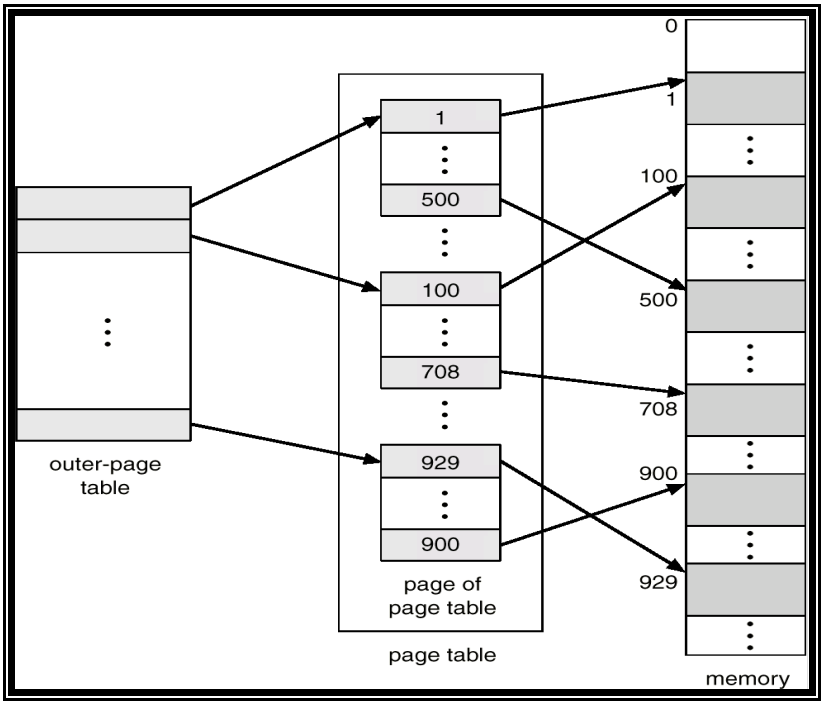

图 5.16 一个两级页表方案

仍以之前一个 4KB 页大小的 32 位系统为例。一个逻辑地址被分为 20 位的页码和 12 位的页偏移。因为要对页表进行再分页，所以该页号可分为 10 位的页码和 10 位的页偏移。这样，x86 两级页表系统就将一个 32 位逻辑地址空间就分为三段，如下图 5.17 的形式：

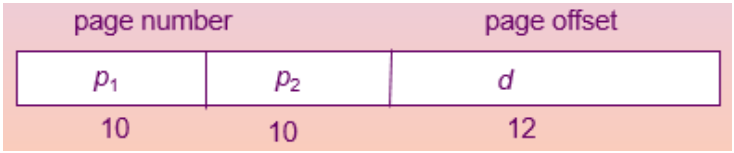

图 5.17 一个两级分页下的逻辑地址

其中，p1 是用来访问外部页表的索引，而 p2 是外部页表的页偏移。每页的大小为 4KB。由于物理块号和页表的物理地址都占 4 个字节，使每页中包含 1024 个页表项，所以页表目录和页表的大小也都是 4KB，即一页。在 x86 中设置了一个外层页表寄存器（CR3），用于存放页表目录的基址。

采用这种结构的地址转换方法如图 5.18 所示。由于地址转换由外向内，这种方案也称为向前映射页表(forward-mapped page table)。

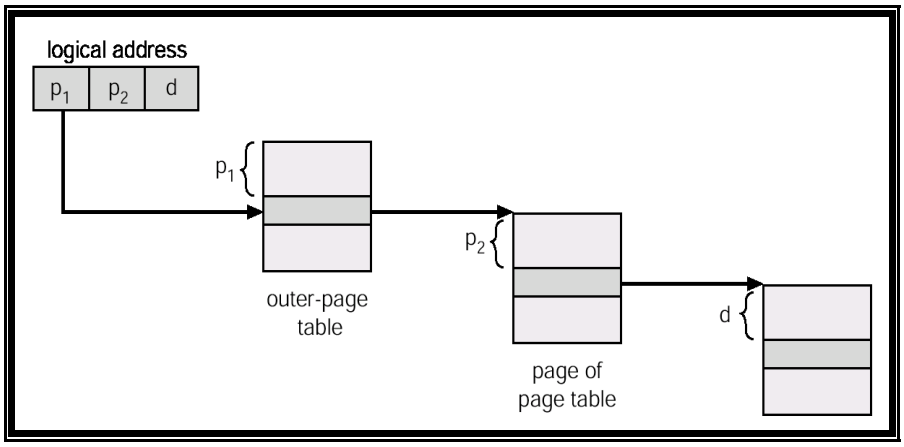

图 5.18 一个两级 32 位分页结构的地址转换机制

(b) 哈希页表

处理超过 32 位地址空间的常用方法是使用哈希页表(hashed page table)，并以虚拟页码作为哈希值。哈希页表的每一条目都包括一个链表的元素，这些元素哈希成同一位置(要处理碰撞)。每个元素有 3 个域：

(1) 虚拟页码

(2) 所映射的帧号

(3) 指向链表中下一个元素的指针。

该算法按如下方式工作：虚拟地址中的虚拟页号转换到哈希表中，用虚拟页号与链表中的每一个元素的第一个域相比较。如果匹配，那么相应的帧号(第二个域)就用来形成物理地址；如果不匹配，那么就对链表中的下一个节点进行比较，以寻找一个匹配的页号。

该方案如图 5.19 所示：

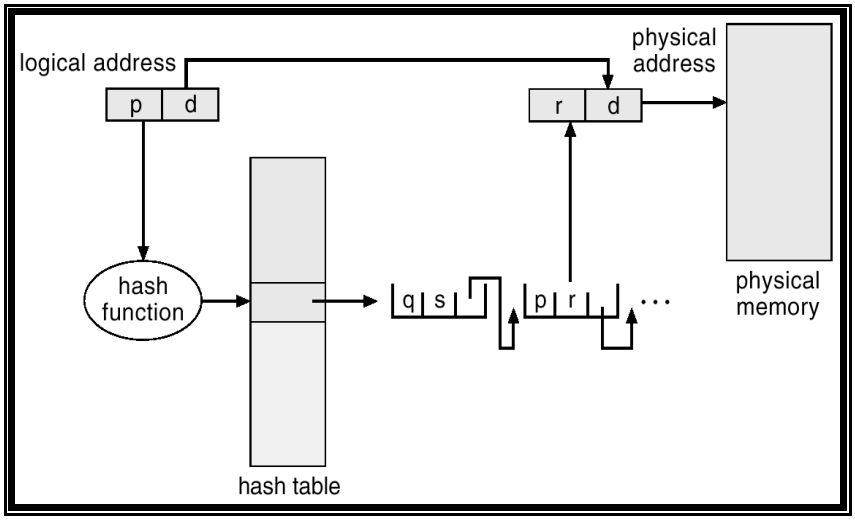

图 5.19 哈希页表

(c) 反向页表

通常，每个进程都有一个相关页表。该进程所使用的每个页都在页表中有一项(或者每个虚拟地址都有一项，不管后者是否有效)。这种页的表示方式比较自然，这是因为进程是通过页的虚拟地址来引用页的。操作系统必须将这种引用转换成物理内存地址。由于页表是按虚拟地址排序的，操作系统能够计算出所对应条目在页表中的位置，并可以直接使用该

值。这种方法的缺点之一是每个页表可能有很多项。这些表可能消耗大量物理内存，却仅用来跟踪物理内存是如何使用的。

为了解决这个问题，可以使用反向页表(inverted page table) 。反向页表不是依据进程的逻辑页号来组织，而是依据该进程在内存中的物理帧号来组织（即：按物理帧号排列）。反向页表对于每个真正的内存页或帧才有一个条目。每个条目包含保存在真正内存位置的页的虚拟地址以及拥有该页的进程的信息。

因此，整个系统只有一个页表，对每个物理内存的页只有一条相应的条目。图 5.20 说明了反向页表的操作，将它与的标准页表操作相比较。因为系统只有一个页表，而有多个地址空间映射物理内存，所以反向页表的条目中通常需要一个地址空间标识符，以确保一个特定进程的一个逻辑页可以映射到相应的物理帧。采用反向页表的系统包括 64 位的 UltraSPARC 和 PowerPC。

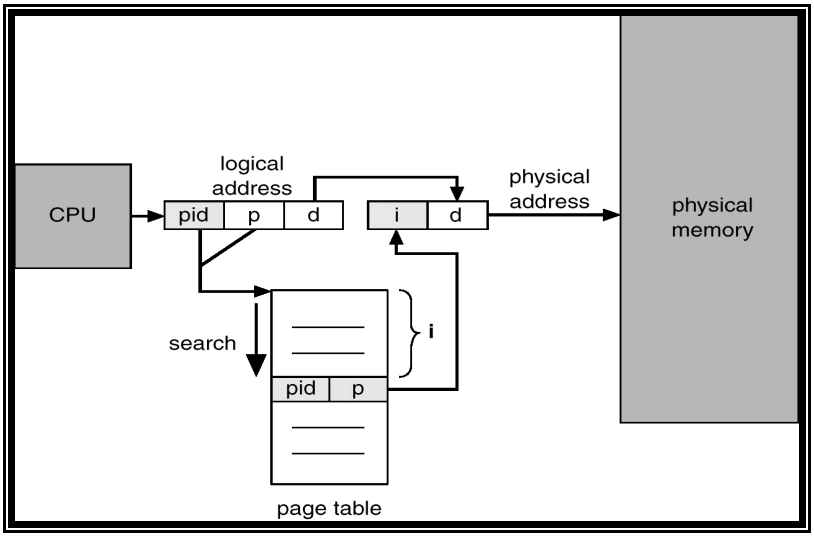

图 5.20 反向页表

为了说明这种方法，这里描述一种简化的反向页表实现，IBM RT 就使用这种方法。系统的每个虚拟地址有一个三元组

<process-id, page-number, offset>

每个反向页表的条目为一对<process-id, page-number, offset>，其中 process-id 用来作为地址空间的标识符。当需要内存引用时，由<process-id, page-number>组成的虚拟地址部分送交内存子系统。通过查找反向页表来寻找匹配。如果匹配找到，例如条目 i，那么就产生了物理地址<i, offset>。如果没有匹配，那么就是试图进行非法地址访问。

虽然这种方案减少了存储每个页表所需要的内存空间，但是当引用页时，它增加了查找页表所需要的时间。由于反向页表按物理地址排序，而查找是根据虚拟地址，因此可能需要查找整个表来寻求匹配。这种查找会花费很长时间。为了解决这一问题，可以使用哈希页表来将查找限制在一个或少数几个页表条目。当然，每次访问哈希页表也为整个过程增加了一次内存引用，因此一次虚拟地址引用至少需要两个内存读：一个查找哈希页表条目，另一个查找页表。为了改善性能，可以在访问哈希页表时先查找 TLB。