第 3 章

虚拟存储器

3.1 概述

在之前的章节中，都认为处理器送出的取指令和取数据的地址是实际的物理地址(physical address)，而现代的高性能处理器都是支持虚拟地址(virtual address)的，为什么需要虚拟地址呢？

在早期人们使用 DOS 或者更古老的操作系统的时候，程序的规模很小，虽然那时候物理内存(Physical Memory)也很小，但这样的物理内存可以容纳下当时的程序，但是随着图形界面的兴起，以及用户需求的不断增大，应用程序的规模也越来越大，于是就有一个难题出现了：应用程序太大以至于物理内存已经无法容纳下这样的程序了，于是通常的做法就是把程序分割成许多的片段，片段 0 首先放到物理内存中执行，当它执行完时调用下一个片段，例如片段 1，虽然片段在物理内存中的交换是操作系统完成的，但是程序员必须先把程序进行分割，这是一个费时费力的工作，相当枯燥，需要更好的方法来解放人力，于是就有了虚拟存储器(Virtual Memory)。它的基本思想是对于一个程序来说，它的程序(code)、数据(data)和堆栈(stack)的总大小可以超过实际物理内存的大小，操作系统把当前使用的部分内容放到物理内存中，而把其他未使用的内容放到更下一级存储器，如硬盘(disk)或闪存(flash)上。举例来说，一个大小为 32MB 的程序运行在物理内存只有 16MB 的机器上，操作系统通过选择，决定各个时刻将程序中一部分 16MB 的内容(或者更小)放在物理内存中，而将其他的内容放到硬盘中，并在需要的时候在物理内存和硬盘之间交换程序片段，这样就可以把大小为 32MB 的程序放到物理内存为 16MB 的机器上运行，而且在运行之前也不需要程序员对程序进行分割。

有了上面的概念，此时就可以说，一个程序是运行在虚拟存储器空间的，它的大小由处理器的位数决定，例如对于一个 32 位处理器来说，其地址范围就是 0～0xFFFF FFFF，也就是 4GB；而对于一个 64 位处理器来说，其地址范围就是 0～0xFFFF FFFF FFFF FFFF，这个范围就是程序能够产生的地址范围，其中的某一个地址就称为虚拟地址。和虚拟存储器相对应的就是物理存储器，它是在现实世界中能够直接使用的存储器，其中的某一个地址就是物理地址。物理存储器的大小不能够超过处理器最大可以寻址的空间，例如，对于 32 位的 x86 PC 来说，它的物理存储器(一般都简称为内存)可以是 256MB，即 PM 的范围是 0～0xFFF FFFF，当然，也可以将物理内存增加到 4GB，此时虚拟存储器和物理存储器这两个地址空间的大小就是相同的，在当前使用的 32 位 x86 PC 中，不可能使用比 4GB 更大

的物理内存了。

在没有使用虚拟地址的系统中，处理器输出的地址会直接送到物理存储器中，这个过程如图 3.1 所示。

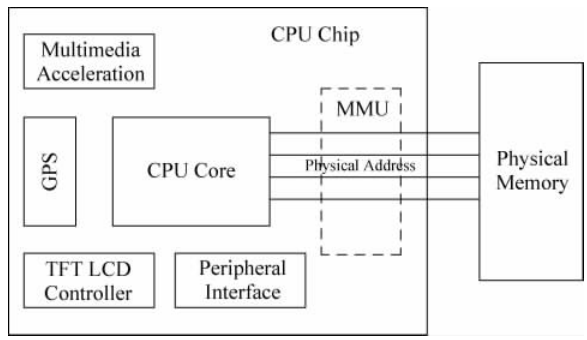

图 3.1　没有使用虚拟存储器的系统

而如果使用了虚拟地址，则处理器输出的地址就是虚拟地址了，这个地址不会被直接送到物理存储器中，而是需要先进行地址转换，因为虚拟地址是没有办法直接寻址物理存储器的，负责地址转换的部件一般称为内存管理单元（Memory Manage Unit，MMU），如图 3.2 所示。

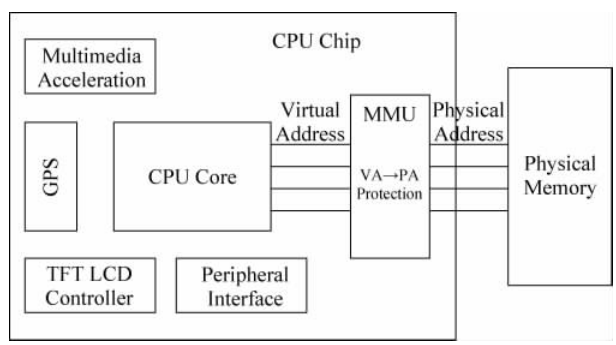

图 3.2　使用虚拟存储器的系统

使用虚拟存储器不仅可以便于程序在处理器中运行，还可以给程序的编写带来好处，在直接使用物理存储器的处理器中，如果要同时运行多个程序，就需要为每个程序都分配一块地址空间，每个程序都需要在这个地址空间内运行，这样极大地限制了程序的编写，而且不能够使处理器随便地运行程序，这样的限制对于应用领域比较专一的嵌入式系统来说，是没有问题的，例如一个 MP3 播放器，事先需要运行什么软件、实现什么功能，都是预先定义好的，因此可以直接使用物理地址；而对于扩展程度很高的复杂系统，例如 PC 或者智能手机，需要能够随意地安装软件，显然是无法事先限制软件运行的地址空间的，这时候就需要虚拟存储器了。使用它之后，每个程序总是认为它占有处理器的所有地址空间，因此程序可以任意使用处理器的地址资源，这样在编写程序的时候，不需要考虑地址的限制，每个程序都认为处理器中只有自己在运行，当这些程序真正放到处理器中运行的时候，由操作系统负责调度，将物理存储器动态地分配给各个程序，将每个程序的虚拟地址转化为相应的物理地址，

使程序能够正常地运行。

通过操作系统动态地将每个程序的虚拟地址转化为物理地址，还可以实现程序的保护，即使两个程序都使用了同一个虚拟地址，它们也会对应到不同的物理地址，因此可以保证每个程序的内容不会被其他的程序随便改写。而且，通过这样的方式，还可以实现程序间的共享，例如操作系统内核提供了打印(printf)函数，第一个程序在地址 A 使用了 printf 函数，第二个程序在地址 B 使用了 printf 函数，操作系统在地址转换的时候，会将地址 A 和 B 都转换为同样的物理地址，这个物理地址就是 printf 函数在物理存储器中的实际地址，这样就实现了程序的共享，虽然两个程序都使用了 printf 函数，但是没必要真的使 printf 函数占用物理存储器的两个地方，因此，使用虚拟存储器不仅可以降低物理存储器的容量需求，它还可以带来另外的好处，如保护(protect)和共享(share)。

可以说，如果一个处理器要支持现代的操作系统，就必须支持虚拟存储器，它是操作系统一个非常重要的内容，本章重点从硬件层面来讲述虚拟存储器，涉及到页表(Page Table)、程序保护和 TLB 等内容。

3.2 地址转换

虚拟存储器从最开始出现一直到现在，有很多种实现的方式，本节只讲述目前最通用的方式——基于分页(page)的虚拟存储器。当前大多数的处理器都使用了这种分页机制，虚拟地址空间的划分以页(page)为单位，典型的页大小为 4KB，相应的物理地址空间也进行同样大小的划分，由于历史的原因，在物理地址空间中不叫做页，而称为 frame，它和页的大小必须相等，例如，它们的典型值都是 4KB。当程序开始运行时，会将当前需要的部分内容从硬盘中搬移到物理内存中，每次搬移的单位就是一个页的大小，因此页和 frame 的大小也就必须相等了。由于只有在需要的时候才将一个页的内容放到物理内存中，这种方式就称为 demand page，它使处理器可以运行比物理内存更大的程序。对于一个虚拟地址 VA 来说，VA[11:0]用来表示页内的位置，称为 page offset，VA 剩余的部分用来表示哪个页，也称为 VPN(Virtual Page Number)。相应的，对于一个物理地址 PA 来说，PA[11:0]用来表示 frame 内的位置，称为 frame offset，而 PA 剩余的部分用来表示哪个 frame，也称为 PFN(Physical Frame Number)。由于页和 frame 的大小是一样的，所以从 VA 到 PA 的转化实际上也就是从 VPN 到 PFN 的转化，offset 的部分是不需要变化的。

下面通过一个例子来说明虚拟地址到物理地址的转化过程。

在图 3.3 所示的例子中，假设处理器是 16 位的，则它的虚拟地址范围是 0～0xFFFF，共 64KB，页的大小是 4KB，因此 64KB 的虚拟地址包括了 16 个页，即 16 个 VPN；而这个系统中物理内存只有 32KB，它包括了 8 个 PFN。现在有一个程序，它的大小大于 32KB，这个程序在执行的时候，不能够一次性调入内存中运行，因此这台机器必须有一个可以存放这个程序的下一级存储器(例如硬盘或者闪存)，以保证程序片段在需要的时候可以被调用。在图 3.3 中，一部分虚拟地址已经被映射到了物理空间，例如 VPN0(地址范围 0～4K)被映射为 PFN2(地址范围 8～12K)；VPN1(地址范围 4～8K)被映射为 PFN0(地址范围 0～4K)等。

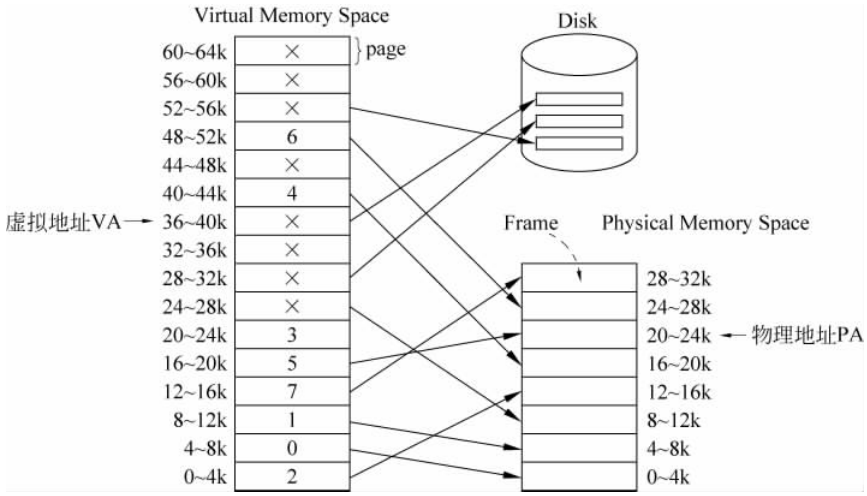

图 3.3　地址转换的一个例子

前面已经说过，在程序中使用的地址都是虚拟地址，虚拟地址会被送到处理器中专门负责地址转换的部件，即 MMU，被转换为物理地址之后，再去访问物理内存而得到真正的数据。为了便于理解，下面以 load 指令为例子进行描述，不过只考虑取数据的过程，其实每条 load 指令自身的地址（或者称为 PC 值）也是虚拟地址，只是为了避免混淆，此时不考虑这个事情。对于从虚拟地址到物理地址的转换来说，只是对 VPN 进行操作，页内的偏移是不需要进行转化的，也就是说，页是进行地址转换的最小单位。例 1：

```
Load R2, 5[R1] ;     //假设 R1 的值为 0
```

这条 load 指令在执行的时候，得到的取数据的虚拟地址是 R1＋5＝5，也就是地址 5 会被送到 MMU 中，从图 3.3 中可以看到，地址 5 落在了 page0 的范围之内（它的范围是 0～4095），而 page0 被映射到物理空间中的 frame2（它的地址范围是 8192～12287），因此 MMU 将虚拟地址 5 转换为物理地址 8192＋5＝8197，并把这个地址送到物理内存中取数据，物理内存并不知道 MMU 做了什么映射，它只是看到了一个对地址 8197 进行读操作的任务。例 2：

```
Load R2, 0[R1] ;     //假设 R1 的值为 20500
```

从图 3.3 中可以看出，用来取数据的虚拟地址 20500 落在了 page5 内（它的地址范围是 20480～24575），和页的起始地址有 20 个字节的距离，而且 page5 被映射到了物理内存中的 frame3（它的地址范围是 12288～16383），由于页内的偏移是不会变化的，因此被映射的物理地址是 12288＋20＝12308。例 3：

```
Load R1, 0[R1] ;     //假设 R1 的值为 32780
```

从图 3.3 中可以看出，虚拟地址 32780 落在了 page8 的范围内，而 page8 并没有一个有效的映射，即此时 page8 的内容没有存在于物理内存中，而是存在于硬盘中。MMU 发现这个页没有被映射之后，就产生一个 Page Fault 的异常送给处理器，这时候处理器就需要转

到 Page Fault 对应的异常处理程序中处理这个事情(这个异常处理程序其实就是操作系统的代码),它必须从物理内存的八个 frame 中找到一个当前很少被使用的,假设选中了 frame1,它和 page2 有着映射关系,所以首先将 frame1 和 page2 解除映射关系,此时虚拟地址空间中 page2 的地址范围就被标记为没有映射的状态,然后把需要的内容(本例中是 page8)从硬盘搬移到物理内存中 frame1 的空间,并将 page8 标记为映射到 frame1;如果这个被替换的 frame1 是脏(dirty)状态的,还需要先将它的内容搬移到硬盘中,这里脏(dirty)的概念和 Cache 中是一样的,表示这个 frame 以前被修改过,被修改的数据还没有来得及更新到硬盘中,关于这部分的详细内容会在后文进行介绍。处理完上述的内容,就可以从 Page Fault 的异常处理程序中进行返回,此时会返回到这条产生 Page Fault 的 load 指令,并重新执行,这时候就不会再产生异常,可以取到需要的数据了。

3.2.1 单级页表

对于处理器来说,当它需要的页(page)不在物理内存中时,就发生了 Page Fault 类型的异常,需要访问更下一级的存储器,如硬盘,而硬盘的访问时间一般是以 ms 为单位的,这需要很长的一段时间,严重降低了处理器的性能,因此应该尽量减少 Page Fault 发生的频率,这就需要优化页在物理内存中的摆放。如果允许一个页能够放到物理内存中任意一个 frame 中,这时候操作系统就可以在发生 Page Fault 的时候,将物理内存中任意一个页进行替换,具备这项功能之后,操作系统可以利用一些比较复杂的算法,将物理内存中那些最近一段时间都没有使用过的页进行替换,这样可以保证不会将一些当前很活跃的内容踢出物理内存,也就是说,使用比较灵活的替换算法能够减少 Page Fault 发生的频率。

但是,这种在物理内存中任意的替换方法(类似于全相连结构的 Cache)直接实现起来不是很容易,所以在使用虚拟存储器的系统中,都是使用一张表格来存储从虚拟地址到物理地址(实际上是 VPN 到 PFN)的对应关系,这个表格称为页表(Page Table,PT),这个表格放在什么地方呢?当然可以在处理器中使用寄存器来存储它,但是考虑到它会占用不菲的硬件资源,很少有人会这样做,一般都是将这个表格放在物理内存中,使用虚拟地址来寻址,表格中被寻址到的内容就是这个虚拟地址对应的物理地址。每个程序都有自己的页表,用来将这个程序中的虚拟地址映射到物理内存中的某个地址,为了指示一个程序的页表在物理内存中的位置,在处理器中一般都会包括一个寄存器,用来存放当前运行程序的页表在物理内存中的起始地址,这个寄存器称为页表寄存器(Page Table Register,PTR),每次操作系统将一个程序调入物理内存中执行的时候,就会将寄存器 PTR 设置好,当然,上面的这种机制可以工作的前提是页表位于物理内存中一片连续的地址空间内。

图 3.4 表示了如何使用 PTR 从物理内存中定位到一个页表,并使用虚拟地址来寻址页表,从而找到对应的物理地址的过程。其实,使用 PTR 和虚拟地址共同来寻址页表,这就相当于使用它们两个共同组成一个地址,使用这个地址来寻址物理内存。图 3.4 中仍然假设每个页的大小是 4KB,使用 PTR 和虚拟地址共同来寻址页表,找到对应的表项(entry),当这个表项对应的有效位(valid)是 1 时,就表示这个虚拟地址所在的 4KB 空间已经被操作系统映射到了物理内存中,可以直接从物理内存中找到这个虚拟地址对应的数据,其实,这时候访问当前页内任意的地址,就是访问物理内存中被映射的那个 4KB 的空间了。相反,如

果页表中这个被寻址的表项对应的有效位是 0，则表示这个虚拟地址对应的 4KB 空间还没有被操作系统映射到物理内存中（为了节省物理内存的使用，操作系统只会把那些当前被使用的页映射到物理内存中），则此时就产生了 Page Fault 类型的异常，需要操作系统从更下一级的存储器中（例如硬盘或者闪存）将这个页对应的 4KB 内容搬移到物理内存中。

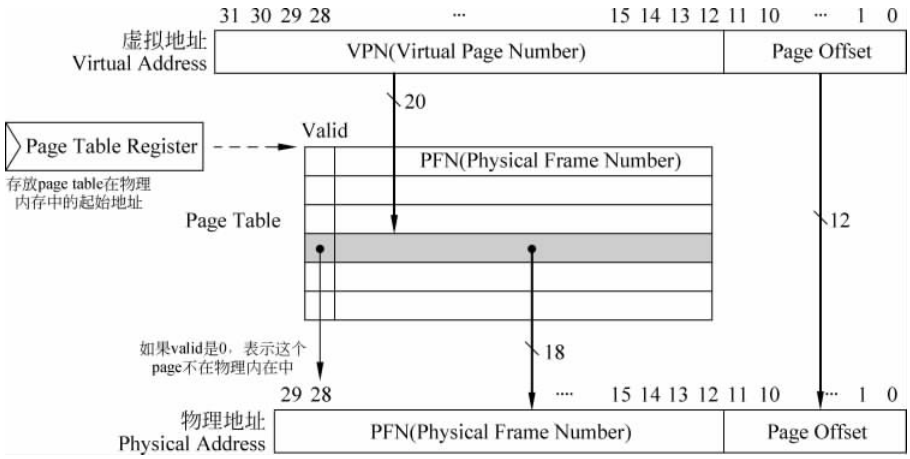

图 3.4　通过页表进行地址转换

在图 3.4 中，使用了 32 位的虚拟地址，页表在物理内存中的起始地址是用 PTR 来指示的。虚拟地址的寻址空间是 2^{32} 字节，也就是 4GB；物理地址的寻址空间是 2^{30} 字节，也就是 1GB，这就像是目前使用的 32 位的 PC，虽然最大支持 4GB 的内存，但是很多时候，计算机上实际安装的内存可能只有 1GB，甚至是 512MB，这就对应着实际物理地址的寻址空间。在页表中的一个表项（entry）能够映射 4KB 的大小，为了能够映射整个 4GB 的空间，需要表项的个数应该是 4GB/4KB=1M，也就是 2^{20}，因此需要 20 位来寻址，也就是虚拟地址中除了 Page Offset 之外的其他部分，这个其实是很自然的，因为 32 位的虚拟地址能够寻址 4GB 的空间，将其人为地分为两部分，低 12 位用来寻址一个页内的内容，高 20 位用来寻址哪个页，也就是说，真正寻址页表的其实不是虚拟地址的所有位数，而只是 VPN 就可以了，从页表中找到的内容也不是整个的物理地址，而只是 PFN。从图 3.4 来看，页表中的每个表项似乎只需要 18 位的 PFN 和 1 位的有效位（valid），也就是总共 19 位就够了。实际上因为页表是放到物理内存中，而物理内存中的数据位宽都是 32 位的，所以导致页表中每个表项的大小也是 32 位的，剩余的位用来表示一些其他的信息，如每个页的属性信息（是否可读或可写）等，这样页表的大小就是 4B×1M = 4MB，也就是说，按照目前的讲述，一个程序在运行的时候，需要在物理内存中划分出 4MB 的连续空间来存储它的页表，然后才可以正常地运行这个程序。

需要注意的是，页表的结构是不同于 Cache 的，在页表中包括了所有 VPN 的映射关系，所以可以直接使用 VPN 对页表进行寻址，而不需要使用 Tag。

在处理器中，一个程序对应的页表，连同 PC 和通用寄存器一起，组成了这个程序的状态，如果在当前程序执行的时候，想要另外一个程序使用这个处理器，就需要将当前程序的状态进行保存，这样就可以在一段时间之后将这个程序进行恢复，从而使这个程序可以继续执行。在操作系统中，通常将这样的程序称为进程（process），当一个进程被处理器执行的

时候，称这个进程是活跃的(active)，否则就称之为不活跃(inactive)。操作系统通过将一个进程的状态加载到处理器中，就可以使这个进程进入活跃的状态。可以说，进程是一个动态的概念，当一个程序只是放在硬盘中，并没有被处理器执行的时候，它只是一个由一条条指令组成的静态文件，只有当这个程序被处理器执行时，例如用户打开了一个程序，此时才有了进程，需要操作系统为其分配物理内存中的空间，创建页表和堆栈等，这时候一个静态的程序就变为了动态的进程，这个进程可能是一个，也可能是多个，这取决于程序本身。当然，进程是有生命期的，一旦用户关闭了这个程序，进程也就不存在了，它所占据的物理内存也会被释放掉。

一个进程的页表指定了它能够在物理内存中访问的地址空间，这个页表当然也是位于物理内存中的，在一个进程进行状态保存的时候，其实并不需要保存整个的页表，只需要将这个页表对应的 PTR 进行保存即可。因为每个进程都拥有全部的虚拟存储器空间，因此不同的进程肯定会存在相同的虚拟地址，操作系统需要负责将这些不同的进程分配到物理内存中不同的地方，并将这些映射(mapping)信息更新到页表中(使用 store 指令就可以完成这个任务)，这样不同的进程使用的物理内存就不会产生冲突了。

图 3.5 表示了在处理器中存在三个进程的情况，每个进程都有自己的页表，虽然在三个进程中都存在相同的虚拟地址 VA1，但是通过每个进程自己的页表，将它们的 VA1 映射为物理内存中不同的物理地址。同理，虽然三个进程中存在不同的虚拟地址 VA2、VA3 和 VA4，但是它们都是访问同一个函数，因此通过每个进程的页表将它们都映射到了物理内存中的同一个地址，通过这种方式，实现不同进程之间的保护和共享。

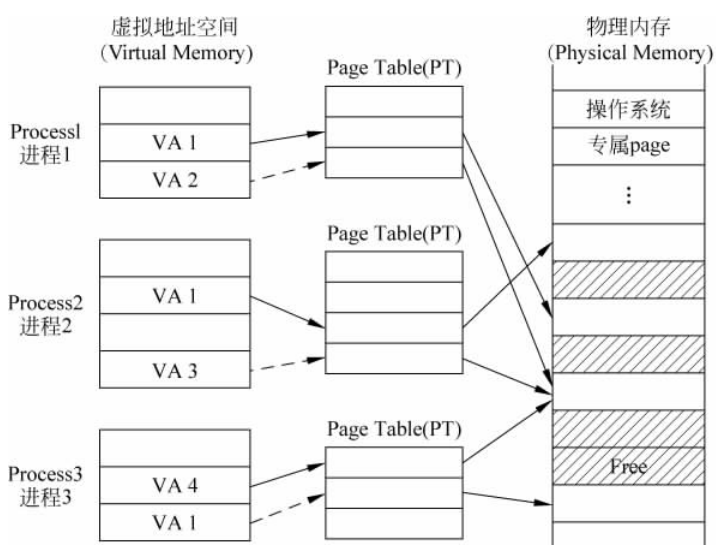

图 3.5　系统中存在三个进程时的地址转换

但是在之前说过，为了节省硅片面积，都会把页表放到物理内存中，这样要得到一个虚拟地址对应的数据，需要访问两次物理内存。第一次访问物理内存中的页表，得到对应的物理地址；第二次使用这个物理地址来访问物理内存，这样才能得到需要的数据，如图 3.6 表示了这个过程。

按照图 3.6 的这种访问过程，要执行一条 load 指令而得到数据，需要的时间肯定会很

长，毕竟物理内存的访问速度和处理器的速度相比是很慢的。可以说，图 3.6 的这种访问数据的方法没有错误，但是效率不高，现实当中的处理器都会使用 TLB 和 Cache 来加快这个过程，这些内容将在后文介绍。

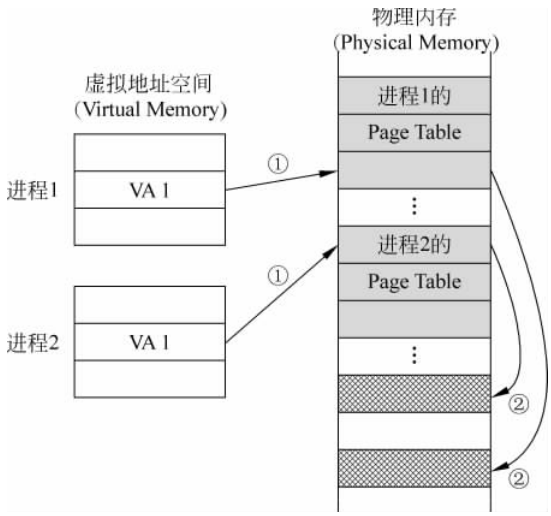

图 3.6　需要访问两次物理内存才可以得到数据

在前面已经计算过，如果一个 32 位处理器中，页的大小是 4KB，并且页表中每个表项的大小是 4 字节的这种典型结构，在页表中需要的表项个数是 4GB/4KB＝2^{20}，所以页表的大小是 4B×2^{20}＝4MB。这也就是说，对于在处理器中运行的每个进程，都需要在物理内存中为其分配连续的 4MB 空间来存储它的页表。当然，如果处理器中只运行一个进程的话，那么看起来问题不大，但是，如果一个处理器中同时运行着上百个进程时（这很常见，打开计算机的任务管理器就可以看到当前有多少进程在运行了），每个进程都要占用 4MB 的物理内存空间来存储页表，那显然就不可能了。而且，对于 64 位的处理器来说，仍旧有可能采用大小为 4KB 的页，可以计算出此时页表需要的空间是非常巨大的，大小是 4B×(2^{64}/2^{12}) ＝ 4B×2^{52}＝ 2^{54}B，这显然已经不可能了。

事实上，一个程序很难用完整个 4GB 的虚拟存储器空间，大部分程序只是用了很少一部分，这就造成了页表中大部分内容其实都是空的，并没有被实际地使用，这样整个页表的利用效率其实是很低的。

可以采用很多方法来减少一个进程的页表对于存储空间的需求，最常用的方法是多级页表（Hierarchical Page Table），这种方法可以减少页表对于物理存储空间的占用，而且非常容易使用硬件来实现，与之对应的，本节所讲述的页表就称为单级页表（Single Page Table），也被称为线性页表（Linear Page Table）。

3.2.2　多级页表

在多级页表（Hierarchical Page Table）的设计中，将 3.2.1 节介绍的一个 4MB 的线性页表划分为若干个更小的页表，称它们为子页表，处理器在执行进程的时候，不需要一下子

把整个线性页表都放入物理内存中，而是根据需求逐步地放入这些子页表。而且，这些子页表不再需要占用连续的物理内存空间了，也就是说，两个相邻的子页表可以放在物理内存中不连续的位置，这样也提高了物理内存的利用效率。但是，由于所有的子页表是不连续地放在物理内存中，所以仍旧需要一个表格，来记录每个子页表在物理内存中存储的位置，称这个表格为第一级页表(Level1 Page Table)，而那些子页表则为第二级页表(Level2 Page Table)，如图 3.7 所示。

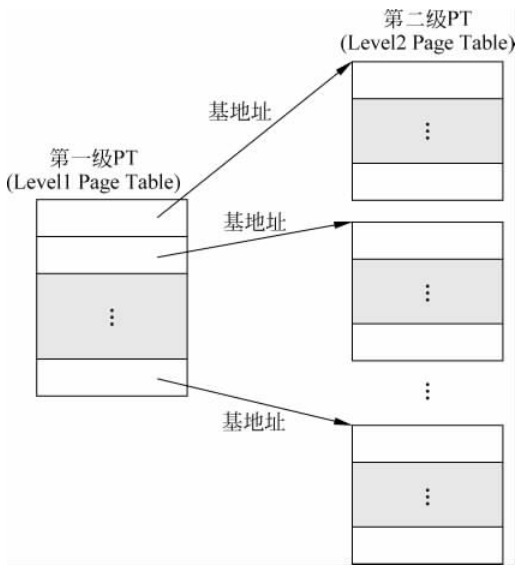

图 3.7　两级页表

这样，要得到一个虚拟地址对应的数据，首先需要访问第一级页表，得到这个虚拟地址所属的第二级页表的基地址，然后再去第二级页表中才可以得到这个虚拟地址对应的物理地址，这时候就可以在物理内存中取出相应的数据了。

举例来说，对于一个 32 位虚拟地址、页大小为 4KB 的系统来说，如果采用线性页表，则页表中的表项个数是 2^{20}，将其分为 1024(2^{10})等份，每个等份就是一个第二级页表，共有 1024 个第二级页表，对应着第一级页表中的 1024 个表项。也就是说，第一级页表需要 10 位地址进行寻址。每个二级页表中，表项的个数是 $2^{20}/2^{10}=2^{10}$个，也需要 10 位地址才能寻址第二级页表，如图 3.8 所示为上述过程的示意图。

在图 3.8 中，一个页表中的表项简称为 PTE(Page Table Entry)，当操作系统创建一个进程时，就在物理内存中为这个进程找到一块连续的 4KB 空间($4B\times 2^{10}=4KB$)，存放这个进程的第一级页表，并且将第一级页表在物理内存中的起始地址放到 PTR 寄存器中。通常这个寄存器都是处理器中的一个特殊寄存器，例如 ARM 中的 TTB 寄存器，x86 中的 CR3 寄存器等。随着这个进程的执行，操作系统会逐步在物理内存中创建第二级页表，每次创建一个第二级页表，操作系统就要将它的起始地址放到第一级页表对应的表项中，如表 3.1 所示为一个进程送出的虚拟地址和第一级页表、第二级页表的对应关系。

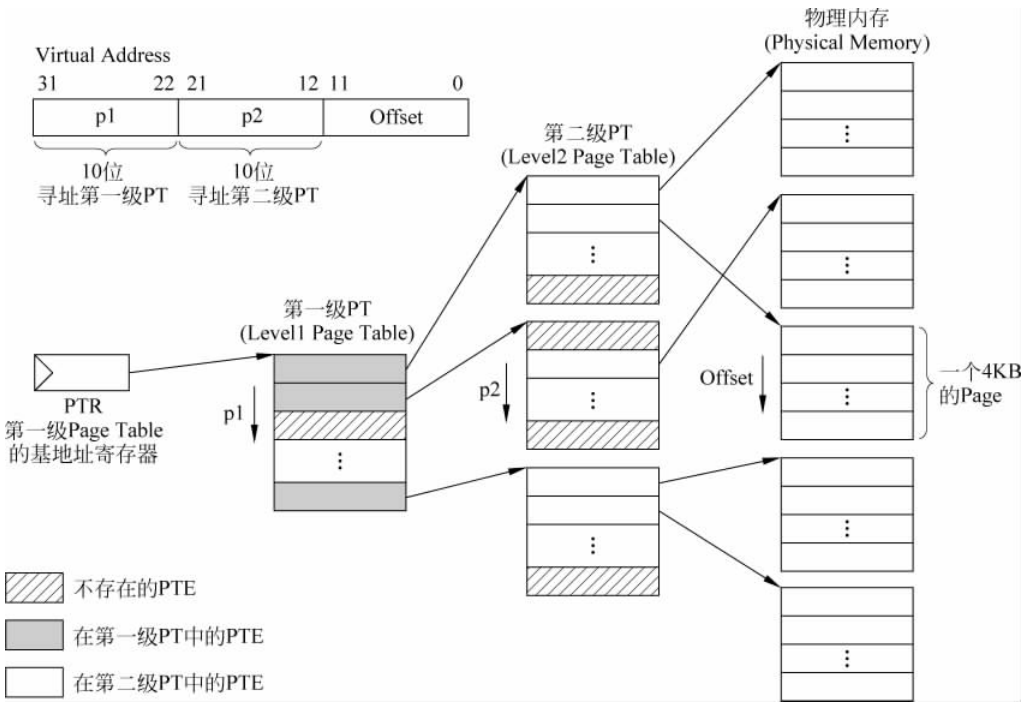

图 3.8　使用两级页表进行地址转换的一个例子

表 3.1　虚拟地址和第一级页表、第二级页表的关系

虚拟地址 VA			Page Table 建立情况	
p1 VA[31:22]	p2 VA[21:12]	Offset VA[11:0]	第二级页表 创建情况	第一级页表 填充情况
0x000	0x000～0x3FF	不关心	创建第 1 个 L2 页表，并根据 p2 的值，逐步填充这个页表	第 1 个 PTE 被填充
0x001	0x000～0x3FF	不关心	创建第 2 个 L2 页表，并根据 p2 的值，逐步填充这个页表	第 2 个 PTE 被填充
…			…	
0x3FF	0x000～0x3FF		创建第 1024 个 L2 页表，并根据 p2 的值，逐步填充这个页表	第 1024 个 PTE 被填充

在图 3.8 所示的系统中，由于虚拟地址(VA)的 p1 部分和 p2 部分的宽度都是 10 位，因此它们的变化范围都是 0x000～0x3FF，每次当虚拟地址的 p1 部分变化时(例如虚拟地址从 0x003FFFFF 变化为 0x00400000，此时 p1 从 0x000 变化为 0x001)，操作系统就需要在物理内存中创建一个新的第二级页表，并将这个页表的起始地址写到第一级页表对应的 PTE 中(由虚拟地址的 p1 部分指定)。当虚拟地址的 p1 部分不发生变化，只是 p2 部分的变化范围在 0x000～0x3FF 之内时，此时不需要创建新的第二级页表。每当虚拟地址中的 p2 部分发生变化，就表示要使用一个新的页，操作系统将这个新的页从下级存储器中(如硬盘)取出来并放到物理内存中，然后将这个页在物理内存中的起始地址填充到第二级页表对应的 PTE

中(由虚拟地址的 p2 部分指定)。至于虚拟地址的页内偏移(即 Offset)部分,只是用来在一个页的内部找到对应的数据,它不会影响第一级和第二级页表的创建。

表 3.1 列出了虚拟地址在所有可能取值的范围内,第一级和第二级页表的创建情况。实际上,绝大部分进程只会使用到全部虚拟地址范围内的一部分地址,例如图 3.8 中,第二级页表中很多的 PTE 都没有被使用,其实这就表示这些 PTE 对应的虚拟地址在进程中是没有出现的。

采用图 3.8 所示的这种分层次的多级页表之后,对于同样大小的程序,如果在程序中使用不同的虚拟地址,会造成第二级页表占用的存储空间有所不同。为什么这样说呢? 举例来说,在一个 32 位虚拟地址、页大小为 4KB 的系统中,有一个大小为 4MB 的程序需要执行,此时需要在物理内存中占用多大的空间来存储页表呢? 这里只考虑程序本身占用的页表,并不考虑在运行过程中的数据占用页表的情况,也就是说,下面所说的虚拟地址指的是取指令时使用的,考虑两种极端的情况。

(1) 最好情况:程序的虚拟地址是连续的,它的变化范围为 0x00000000～0x003FFFFF,共有 4MB 的空间,由于虚拟地址的高 10 位(即 p1 部分)一直是 0x000,因此只需要占用第一级页表中的 PTE_0;虚拟地址的中间 10 位(即 p2 部分)的变化范围为 0x000～0x3FF,正好占用了一个第二级页表中的全部 PTE(PTE_0～PTE_{1023}),所以,这个大小为 4MB 的程序需要使用第一级页表的 PTE_0 以及一个完整的第二级页表。其实在处理器中,为了便于查找,第一级页表总是全部放在物理内存中,因此这个程序所占用的物理内存空间是:

一个第一级页表(必须有)+ 一个第二级页表 = 8KB

(2) 最坏情况:4MB 大小的程序,它的虚拟地址不是连续的,每一个 4KB 的范围都分布在 4MB 的边界之内,如图 3.9 所示。

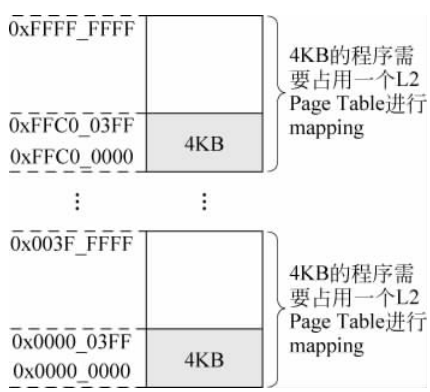

图 3.9 在程序中,每 4KB 的内容分布在 4MB 的边界之内

当程序中出现很多跳转时,如图 3.9 所示的情况就有可能发生,这个程序的虚拟地址的取值就是 0x00000000～0x000003FF、0x00400000～0x004003FF、0x00800000～0x008003FF…,也就是在整个 4GB 的虚拟存储器空间中,每 4MB 大小的空间内,分布着 4KB 大小的程序,整个 4MB 大小的程序需要分布在虚拟存储器中 1024 个 4MB 的区域内。根据表 3.1 可以知道,虚拟地址中的每一个连续的 4MB 区域,都需要一个完整的第二级页表来进行映射(第二级页表中的每个 PTE 都可以映射一个 4KB 的页,所以一个有着 1024 个 PTE 的第二级页表可以映射 4MB 的地址范围)。因此,当一个大小为 4MB 的程序按照图 3.9 的方式分布

时，这个程序的页表需要占用的物理存储空间是：

一个第一级页表(必须有)＋ 1024 个第二级页表 ＝ 1025 ×4KB ＝ 4100KB

也就是说，这个大小为 4MB 的程序，也需要将近 4MB 的物理存储空间来存储它的页表。

在这种极端的情况下，这个 4MB 大小的程序一直分布在了整个 4GB 的虚拟存储器空间内，虽然对于这个程序来说，确实整个 4GB 的虚拟存储器都是可以使用的，但是很显然在实际当中，没有编译器会这样编译程序，大部分程序在编译的时候，所占用的虚拟存储器的地址范围还是尽量集中的，尽管有整个 4GB 的虚拟存储器空间可以使用，但是一个程序，尤其是本例中这样的小程序，还是会尽量集中地放置在虚拟存储器的某个区间内，因此程序的页表所占用的物理存储空间也不会很多，例如最好的那种情况，每个大小为 4KB 的页只使用了第二级页表中的一个 PTE，大小仅为 4B，因此额外的开销也就是 4B/4KB＝0.1% 而已。

总体来看，当处理器开始执行一个程序时，就会把第一级页表放到物理内存中，直到这个程序被关闭为止，因此第一级页表所占用的 4KB 存储空间是不可避免的，而第二级页表是否在物理内存当中，则是根据一个程序当中虚拟地址的值来决定的，操作系统会逐个地创建第二级页表，这个概念和前面讲述过的 Demand Page 是类似的。事实上，伴随着一个页被放入到物理内存中，必然会有第二级页表中的一个 PTE 被建立，这个 PTE 会被写入该页在物理内存中的起始地址，如果这个页对应的第二级页表还不存在，那么就需要操作系统建立一个新的第二级页表，这个过程是很自然的。

这种多级页表的结构比较简单，容易用硬件实现页表的查找，因此在很多硬件实现 Page Table Walk 的处理器中，都是采用了这种结构，如 ARM、x86 和 PowerPC 等。所谓的 Page Table Walk 是指当发生 TLB 缺失时，需要从页表中找到对应的映射关系并将其写回到 TLB 的过程，这些内容在后文会进行详细介绍。这种多级页表还有一个优点，那就是它容易扩展，例如当处理器的位数增加时，可以通过增加级数的方式来减少页表对于物理内存的占用，如图 3.10 所示。

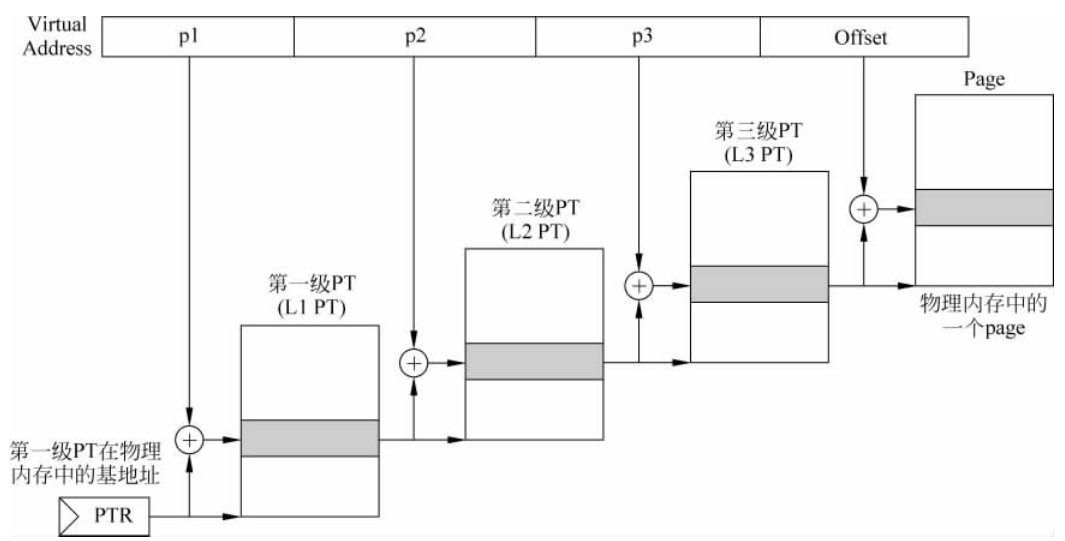

图 3.10　使用多级页表进行地址转换

在这种多级页表的结构中，仍然需要使用一个寄存器来存储第一级页表在物理内存中的基地址，如图 3.10 中的 PTR 寄存器。

使用这种多级页表的结构，每一级的页表都需要存储在物理内存中，因此要得到一个虚拟地址对应的数据，需要多次访问物理内存，很显然，这个过程消耗的时间是很长的，对于一个两级页表来说，图 3.11 表示了两个进程使用虚拟地址得到对应数据的过程。

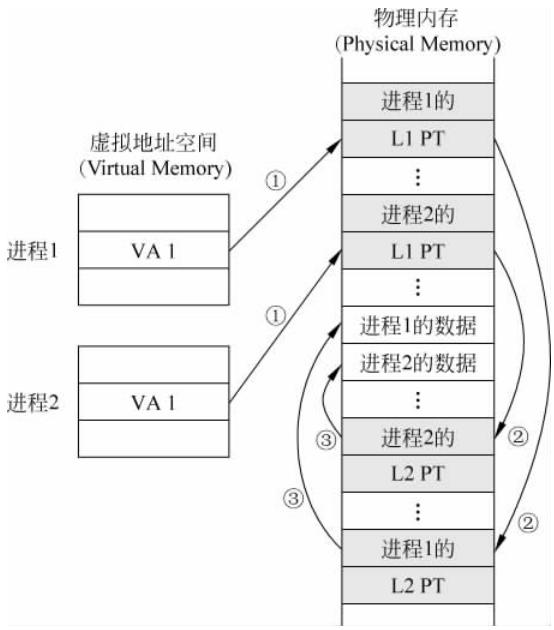

图 3.11　使用两级页表后，需要访问三次物理内存才可以得到数据

对于每个进程来说，需要访问两次物理内存，才能够得到虚拟地址对应的物理地址，然后还需要访问一次物理内存来得到数据，因此要得到虚拟地址对应的数据，共需要访问三次物理内存，如果采用更多级数的页表，所需要的次数还会更多。

到目前为止，已经对虚拟存储器的工作原理进行了介绍，现在对使用虚拟存储器的优点进行总结。

(1) 让每个程序都有独立的地址空间，例如在 32 位的处理器中，每个程序都认为自己拥有整个 4GB 的地址空间，如果不使用虚拟存储器，而是直接使用物理内存，则需要为每个程序都分配一个地址范围，在编写程序时需要限制在这个地址范围之内，这带来了很多不便。

(2) 引入虚拟地址到物理地址的映射，为物理内存的管理带来了方便，可以更灵活地对其进行分配和释放，在虚拟存储器上连续的地址空间可以映射到物理内存上不连续的空间。例如要申请(malloc)一块很大的物理内存空间时，虽然此时在物理内存中有足够的空间，但是却没有足够大的连续空闲内存，此时就可以在物理内存中占用多个不连续的物理页面，将其映射为连续的虚拟地址空间，如图 3.12 所示。使用这种地址映射机制，可以减少物理内存中的碎片，最大限度地利用容量有限的物理内存。

(3) 在处理器中如果存在多个进程，为这些进程分配的物理内存之和可能大于实际可用的物理内存，虚拟存储器的管理使得这种情况下各个进程仍能够正常运行，此时为各个进

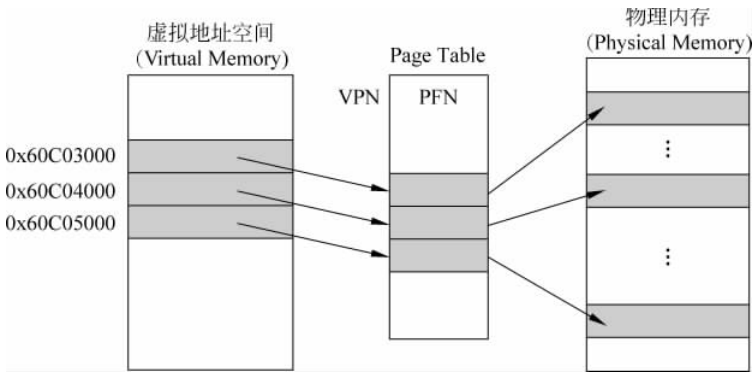

图 3.12　将物理内存中不连续的空间映射为虚拟存储器中连续的空间

程分配的只是虚拟存储器的页，这些页有可能存在于物理内存中，也可能临时存在于更下一级的硬盘中，在硬盘中这部分空间称为 swap 空间。当物理内存不够用时，将物理内存中的一些不常用的页保存到硬盘上的 swap 空间，而需要用到这些页时，再将其从硬盘的 swap 空间加载到物理内存，因此，处理器中等效可以使用的物理内存的总量是物理内存的大小 + 硬盘中 swap 空间的大小。

将一个页从物理内存中写到硬盘的 swap 空间的过程称为 Page Out，将一个页从硬盘的 swap 空间放回到物理内存的过程称为 Page In，如图 3.13 所示为这两个过程的示意图。

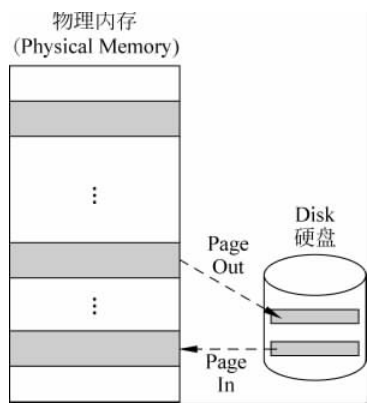

图 3.13　Page In 和 Page Out

(4) 利用虚拟存储器，可以管理每一个页的访问权限，从硬件的角度来看，单纯的物理内存本身不具有各种权限的属性，它的任何地址都可以被读写，而操作系统则要求在物理内存中实现不同的访问权限，例如一个进程的代码段(text)一般不能够被修改，这样可以防止程序错误地修改自己，因此它的属性就要是可读可执行(r/x)，但是不能够被写入；而一个进程的数据段(data)要求是可读可写的(r/w)；同时用户进程不能访问属于内核的地址空间。这些权限的管理就是通过页表(Page Table)来实现的，通过在页表中设置每个页的属性，操作系统和内存管理单元(MMU)可以控制每个页的访问权限，这样就实现了程序的权限管理。

3.2.3 Page Fault

到目前为止，都是使用页表将虚拟地址转换为物理地址，如果一个进程中的虚拟地址在访问页表时，发现对应的 PTE 中，有效位(valid)为 0，这就表示这个虚拟地址所属的页还没有被放到物理内存中，因此在页表中就没有存储这个页的映射关系，这时候就说发生了 Page Fault，需要从下级存储器，例如硬盘中，将这个页取出来，放到物理内存中，并将这个页在物理内存中的起始地址写到页表中。Page Fault 是异常(exception)的一种，通常它的处理过程不是由硬件完成，而是由软件来完成的(确切地说，是由操作系统来完成的，不要忘了操作系统也是软件)，其原因有二。

(1) 由于 Page Fault 时要访问硬盘，这个过程需要的时间很长，通常是毫秒级别，和这个“漫长”的时间相比，即使 Page Fault 对应的异常处理程序需要使用几百条指令，这个时间相比于硬盘的访问时间也是微乎其微的，因此没必要使用硬件来处理 Page Fault。

(2) 发生 Page Fault 时，需要从硬盘中搬移一个或几个页到物理内存中，当物理内存中没有空余的空间时，就需要从其中找到一个最近不经常被使用的页，将其进行替换，使用软件可以根据实际情况实现灵活的替换算法，找到最合适的页进行替换。如果使用硬件的话，很难实现复杂的替换算法，而且不能够根据实际情况进行调整，缺少灵活性。

所以，在现代的处理器中，都是使用软件来处理 Page Fault 的，一旦虚拟地址在访问页表时，发现对应的 PTE 还没有保存相应的映射关系(也就是这个 PTE 的有效位是 0)，那么此时硬件就会产生一个 Page Fault 类型的异常，处理器会跳转到这个异常处理程序的入口地址，异常处理程序会根据某种替换算法，从物理内存中找到一个空闲的地方，将需要页从硬盘中搬移进来，并将这个新的对应关系写到页表中相应的 PTE 内。

需要注意的是，直接使用虚拟地址并不能知道一个页位于硬盘的哪个位置，也需要一种机制来记录一个进程的每个页位于硬盘中的位置。通常，操作系统会在硬盘中为一个进程的所有页开辟一块空间，这就是之前说过的 swap 空间，在这个空间中存储一个进程所有的页，操作系统在开辟 swap 空间的同时，还会使用一个表格来记录每个页在硬盘中存储的位置，这个表格的结构其实和页表是一样的，它可以单独存在，从理论上来讲当然也可以和页表合并在一起，如图 3.14 所示[2]。

如图 3.14 所示在一个页表内记录了一个进程中的每个页在物理内存或在硬盘中的位置，当页表中某个 PTE 的有效位(valid)为 1 时，就表示它对应的页在物理内存中，访问这个页不会发生 Page Fault；相反，如果有效位是 0，则表示它对应的页位于硬盘，访问这个页就会发生 Page Fault，此时操作系统需要从硬盘中将这个页搬移到物理内存中，并将这个页在物理内存中的起始地址更新到页表中对应的 PTE 内。

虽然从图 3.14 看来，映射到物理内存的页表和映射到硬盘的页表可以放到一起，但是在实际当中，物理上它们仍然是分开放置的，因为不管一个页是不是在物理内存中，操作系统都必须记录一个进程的所有页在硬盘中的位置，因此需要单独地使用一个表格来记录它。

在前文已经说过，物理内存相当于是硬盘的 Cache，因为对一个程序来说，它的所有内容其实都存在于硬盘中，只有最近被使用的一部分内容存在于物理内存中，这样符合 Cache 的特征。因为一个程序的某些内容既存在于硬盘中，又存在于物理内存中，当物理内存中某个地址的内容被改变时(例如执行了一条 store 指令，改变了程序中的某个变量)，对于这个

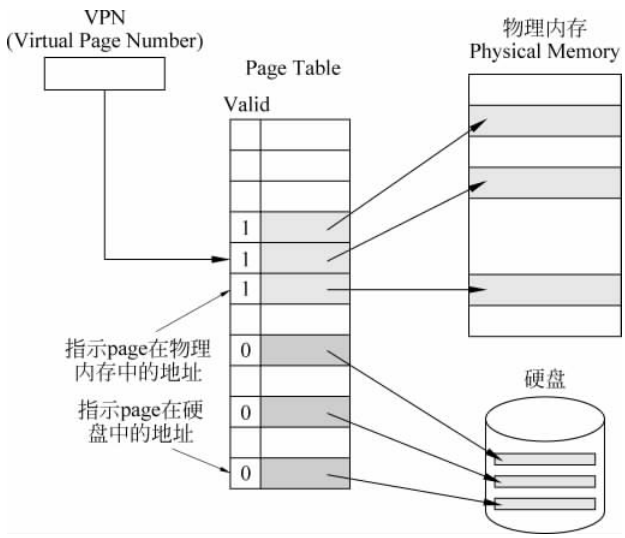

图 3.14　同一个页表中记录了所有的内容

地址来说，在硬盘中存储的内容就过时了，这种情况在 Cache 中也出现过，有两种处理方法。

(1) 写通(Write Through)，将这个改变的内容马上写回硬盘中，考虑到硬盘的访问时间非常慢，这样的做法是不现实的。

(2) 写回(Write Back)，只有等到这个地址的内容在物理内存中要被替换时，才将这个内容写回到硬盘中，这种方式减少了硬盘的访问次数，因此被广泛地使用。

其实，写通(Write Through)的方式只可能在 L1 Cache 和 L2 Cache 之间使用，因为 L2 Cache 的访问时间在一个可以接受的范围之内，而且这样可以降低 Cache 一致性的管理难度，但是更下层的存储器需要的访问时间越来越长，因此只有写回(Write Back)方式才是可以接受的方法。

既然在虚拟存储器的系统中采用了写回的方式，当发生 Page Fault 并且物理内存中已经没有空间时，操作系统需要从其中找到一个页进行替换，如果这个页中的某些内容被修改过，那么在覆盖这个页之前，需要先将它的内容写回到硬盘中，然后才能进行覆盖。为了支持这个功能，需要记录每个页是否在物理内存中被修改过，通常是在页表的每个 PTE 中增加一个脏(dirty)的状态位，当一个页内的某个地址被写入时，这个脏的状态位会被置为 1。当操作系统需要将一个页进行替换之前，会首先去页表中检查它对应 PTE 的脏状态位，如果为 1，则需要先将这个页的内容写回到硬盘中；如果为 0，则表示这个页从来没有被修改过，那么就可以直接将其覆盖了，因为在硬盘中还保存着这个页的内容。从一个时间点来看物理内存，会存在很多的页处于脏的状态，这些页都被写入了新的内容，当然，一旦这些脏的页被写回到硬盘中，它们在物理内存中也就不再是脏的状态了。

由于操作系统还需要在发生 Page Fault 的时候，从物理内存中找到一个页进行替换(当物理内存没有空闲的空间时)，这就需要操作系统实现替换算法，以便能够找到一个最近不经常被使用的页(最近很少被使用，就可以推测它在最近的将来也是一样)。操作系统可以使用 LRU(Least Recently Used)算法进行替换，但是要达到这样的功能，操作系统要使用复杂的数据结构才可以精确地记录物理内存中哪些页最近被使用，这样的代价是很大的：

每次执行访问存储器的指令，都需要操作系统更新这个数据结构。为了帮助操作系统实现这个功能，需要处理器在硬件层面上提供支持，这可以在页表的每个 PTE 中增加一位，用来记录每个页最近是否被访问过，这一位称为“使用位(use)”，当一个页被访问时，“使用位”被置为 1，操作系统周期性地将这一位清零，然后过一段时间再去查看它，这样就能够知道每个页在这段时间是否被访问过，那些最近这段时间没有被使用过的页就可以被替换了。这种方式是近似的 LRU 算法，被大多数操作系统所使用，由于使用了硬件来实现“使用位”，所以操作系统的任务量被大大地减轻了。

总结来说，为了处理 Page Fault，处理器在硬件上需要提供的支持有如下几种。

(1) 在发现 Page Fault 时，能够产生对应类型的异常，并且能够跳转到它的异常处理程序的入口地址。

(2) 当要写物理内存时，例如执行了 store 指令，需要硬件将页表中对应 PTE 的脏状态位置为 1。

(3) 当访问物理内存时，例如执行了 load/store 指令，需要硬件将页表中对应 PTE 的“使用位”置为 1，表示这个页最近被访问过。

需要注意的是，在写回(Write Back)类型的 Cache 中，load/store 指令在执行的时候，只会对 D-Cache 起作用，对物理内存中页表的更新可能会有延迟，当操作系统需要查询页表中的这些状态位时，首先需要将 D-Cache 中的内容更新到物理内存中，这样才能够使用到页表中正确的状态位。

到目前为止，页表中每个 PTE 的内容如图 3.15 所示。

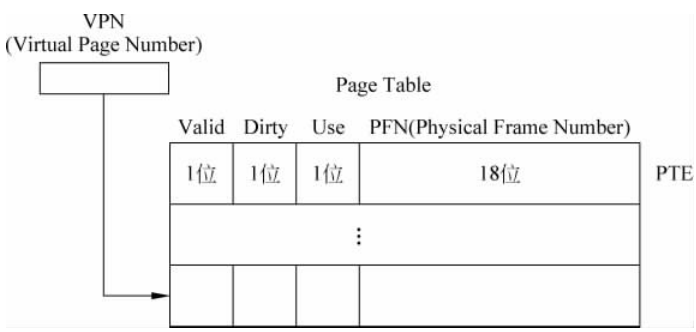

图 3.15　页表中包括的内容

3.2.4　小结

本节讲述了在虚拟存储器的系统中，如何将虚拟地址转换成为物理地址，并介绍了 Page Fault 发生时如何进行处理，在处理器中，有一个模块专门负责虚实地址转换，并且处理 Page Fault，这个模块就是之前介绍的内存管理单元(MMU)，所有支持虚拟存储器的处理器中都会有 MMU 这个模块，下面以单级页表为例，对目前为止讲述的内容进行一个小结。

(1) 当没有 Page Fault 发生时，整个过程如图 3.16 所示。

具体的过程如下。

① 处理器送出的虚拟地址(VA)首先送到 MMU 中。

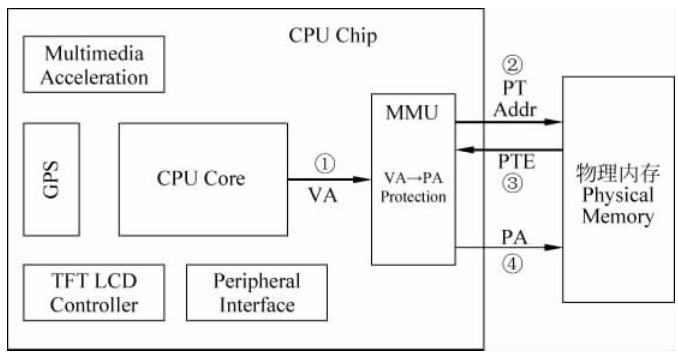

图 3.16　不发生 Page Fault 时的访问流程

② MMU 使用页表的基址寄存器 PTR 和 VA[31:12]组成一个访问页表的地址，这个地址被送到物理内存中。

③ 物理内存将页表中被寻址到的 PTE 返回给 MMU。

④ MMU 判断 PTE 中的有效位，发现其为 1，也就表示对应的页存在于物理内存中，因此使用 PTE 中的 PFN 和原来虚拟地址的[11:0]组成实际的物理地址，即 PA={PFN, VA[11:0]}，并用这个地址来寻址物理内存，得到最终需要的数据。

也就是说，在使用单级页表的情况下，要得到一个虚拟地址对应的数据，需要访问两次物理内存。

(2) 当发生 Page Fault 时，整个过程如图 3.17 所示。

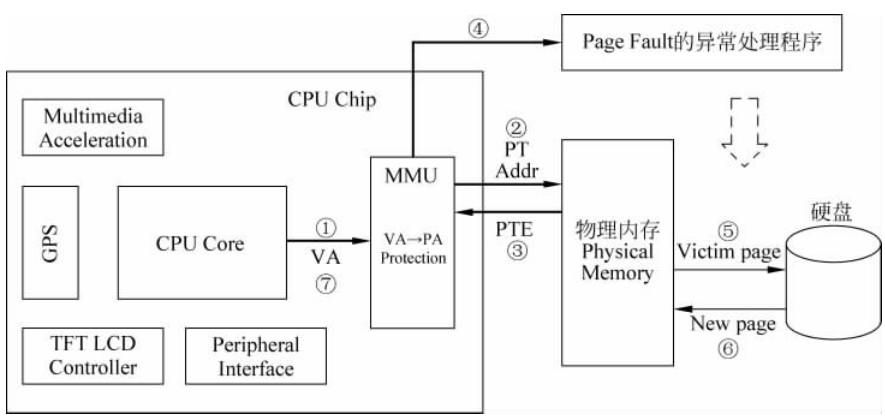

图 3.17　发生 Page Fault 时的访问流程

具体的过程如下。

①～③ 这三个步骤和上面的情况是一样的，处理器送出虚拟地址到 MMU，MMU 使用 PTR 和 VA[31:12]组成访问页表的地址，从物理内存中得到对应的 PTE，送回给 MMU。

④ MMU 查看 PTE 当中的有效位，发现其为 0，也就是需要的页此时不在物理内存当中，此时 MMU 会触发一个 Page Fault 类型的异常送给处理器，这会使处理器跳转到 Page Fault 对应的异常处理程序中，在这一步，MMU 还会把发生 Page Fault 的虚拟地址 VA 也保存到一个专用的寄存器中，供异常处理程序使用。

⑤ 假设此时物理内存中已经没有空闲的空间了，那么 Page Fault 的异常处理程序需要按照某种算法，从物理内存中找出一个未来可能不被使用的页，将其替换，这个页称为 Victim page，如果这个页对应的脏状态位是 1，表示这个页中的内容在以前曾经被修改过，因此需要首先将它从物理内存写到硬盘中，当然，如果脏状态位是 0，那么就不需要写回硬盘这个过程了。

⑥ Page Fault 的异常处理程序会使用 MMU 刚才保存的虚拟地址 VA 来寻址硬盘，找到对应的页，将其写到物理内存中 Victim page 所在的位置，并将这个新的映射关系写到页表中，这里需要注意的是，寻址硬盘的时间是很长的，通常是毫秒级别，因此这一步的处理时间也是很长的。

⑦ 从 Page Fault 的异常处理程序中返回的时候，那条引起 Page Fault 的指令会被重新取到流水线中执行，此时处理器会重新发出虚拟地址送到 MMU 中，因为所需要的页已经被放到物理内存中了，因此这次访问肯定不会发生 Page Fault，会按照不发生 Page Fault 时的过程进行处理。

3.3 程序保护

在现代处理器运行的一个典型环境中，存在着操作系统和许多的用户进程，操作系统的内容如果能够被用户进程随便地修改，那么肯定就会引起系统的崩溃，所以操作系统的内容对于用户进程来说，肯定是不允许被修改的，但是操作系统可能有一部分内容允许用户进程进行读取，例如前文中提到的 printf 函数，由操作系统提供给用户进程来使用；而操作系统相对于普通用户进程来说，应该有足够的权限，以保证操作系统对于整个系统的控制权；不同的进程之间应该也加以保护，一个进程不能让其他的进程随便修改自己的内容，以保证各个进程之间运行的稳定。如果不使用虚拟存储器，要同时满足这些要求是很困难的。

要满足上面的这些需求，需要操作系统和用户进程对于不同的页有不同的访问权限，这一点应该在硬件上就加以控制，通常这种控制是通过页表来实现的，因为要访问存储器的内容，必须要经过页表，所以在页表中对各个页规定不同的访问权限是很自然的事情。需要注意的是，操作系统本身也需要指令和数据，但是考虑到它需要能够访问物理内存中所有的空间，所以操作系统一般不会使用页表，而是直接可以访问物理内存，在物理内存中有一部分地址范围专门供操作系统使用，不允许别的进程随便地访问它，例如在 32 位的 MIPS 处理器中，将整个 4GB 的虚拟存储空间分为了 kseg0、kseg1、kseg2 和 kuseg 共四个区域[11]，其中 kseg0 区域的属性是 unmapped，也就是说，这部分地址是不需要经过页表进行转换的，操作系统内核的指令和数据就是位于 kseg0 区域，而用户进程只能够使用 kuseg 区域，操作系统会根据实际情况，将 kuseg 范围内的虚拟地址映射到物理内存对应的部分，所以事实上，其他的进程也不可能修改物理内存中操作系统专属的“特权区域”，因为操作系统是不会让这种事情发生的。

除了 MIPS，在现实当中的其他处理器也有自己的权限管理方法，举例来说，ARM 处理器也采用了两级页表的方法，第二级页表的每个 PTE 中都有一个 AP 部分，在 ARMv7 架构中，AP 部分直接决定了每个页的访问权限，如表 3.2 所示[12]。

表 3.2 ARM 中每个页的权限管理

AP[2]	AP[1:0]	Privileged 模式访问权限	User 模式访问权限	注 释
0	00	No access	No access	任何模式下,读写该页都会产生异常
0	01	Read/write	No access	该页只允许在 Privileged 模式下进行读写
0	10	Read/write	Read-only	在 User 模式下,写该页会产生异常
0	11	Read/write	Read/write	任何模式下都可以读写该页
1	00	—	—	保留
1	01	Read-only	No access	只允许在 Privileged 模式下读该页
1	10	Read-only	Read-only	只允许在 Privileged 和 User 模式下读该页
1	11	Read-only	Read-only	只允许在 Privileged 和 User 模式下读该页

在 ARMv7 架构中,规定处理器可以工作在 User 模式和 Privileged 模式,在 Privileged 模式下,可以访问处理器内部所有的资源,因此操作系统会运行在这种模式下,而普通的用户程序则是运行在 User 模式下,在表 3.2 中,AP[2:0]位于 PTE 中,通过它,第二级页表可以控制每个页的访问权限,这样可以使一个页对于不同的进程有着不同的访问权限。

既然在页表中规定了每个页的访问权限,那么一旦发现当前的访问不符合规定,例如一个页不允许用户进程访问,但是当前的用户进程却要读取这个页内的某个地址,这样就发生了非法的访问,会产生一个异常(exception)来通知处理器,使处理器跳转到异常处理程序中,这个处理程序一般是操作系统的一部分,由操作系统决定如何处理这种非法的访问,例如操作系统可以终止当前的用户进程,以防止恶意的程序对系统造成破坏,图 3.18 表示了在地址转换的过程中加入权限检查的过程,注意此时在 PTE 中多了用来进行权限控制的 AP 部分。

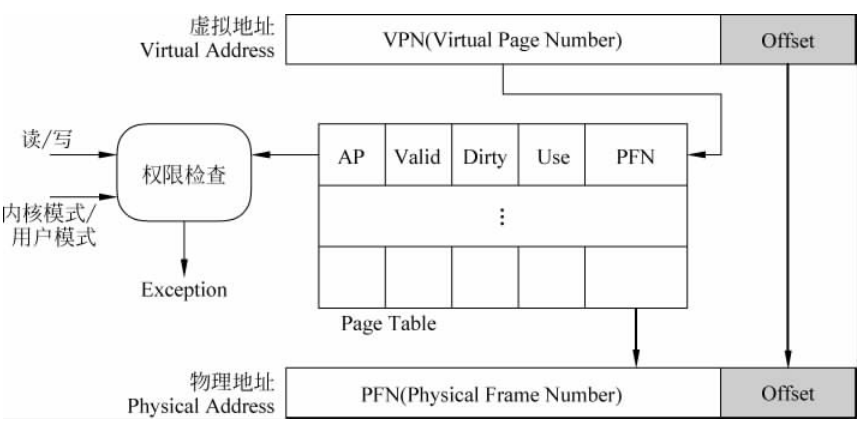

图 3.18 加入程序保护之后的地址转换

当然,如果采用了两级页表的结构,那么图 3.18 只给出了第二级页表的工作过程,事实上在第一级页表中也可以进行权限控制,而且可以控制更大的地址范围。举例来说,在前文讲述的页大小为 4KB 的系统中,第一级页表中每个 PTE 都可以映射一个完整的第二级页表,也就是 4KB×1024=4MB 的地址范围。也就是说,第一级页表中的每个 PTE 都可以控制 4MB 的地址范围,这样可以更高效地对大片的地址进行权限设置和检查。例如,可以在

第一级页表的每个 PTE 中设置一个两位的权限控制位，当其为 00 时，它对应的整个 4MB 空间都不允许被访问；当为 11 时，对应的 4MB 空间将不设限制，随便访问；当为 01 时，需要查看第二级页表的 PTE，以获得关于每个页自身访问权限的情况；通过这种粗粒度（第一级页表的权限控制）和细粒度（第二级页表的权限控制）的组合，可以在一定程度上提高处理器的执行效率。

在后文会讲到，存在 D-Cache 的系统中，处理器送出的虚拟地址经过页表转化为物理地址之后，其实并不会直接去访问物理内存，而是先访问 D-Cache。如果是读操作，那么直接从 D-Cache 中读取数据；如果是写操作，那么也是直接写 D-Cache（假设命中），并将被写入的 Cache line 置为脏（dirty）的状态。只要 D-Cache 是命中的，就不需要再去访问物理内存了。但是，如果处理器送出的虚拟地址并不是要访问物理内存，而是要访问芯片内的外设寄存器，例如要访问 LCD 驱动模块的寄存器，此时对这些寄存器的读写是为了对外设进行操作，因此这些地址是不允许经过 D-Cache 被缓存的，如果被缓存了，那么这些读写将只会在 D-Cache 中起作用，并不会传递到外设寄存器中而真正对外设模块进行操作，这样显然是不可以的，因此在处理器的存储器映射（memory map）中，总会有一块区域，是不可以被缓存的。例如 MIPS 处理器的 kseg1 区域就不允许被缓存，它的属性是 uncached，这个属性也应该在页表中加以标记，在访问页表从而得到物理地址时，会对这个地址对应的页是否允许缓存进行检查，如果发现这个页的属性是不允许缓存的，那么就需要直接使用刚刚得到的物理地址来访问外设或者物理内存；如果这个页的属性是允许被缓存，那么就可以直接使用物理地址对 D-Cache 进行寻址。

到目前为止，总结起来，在页表中的每个 PTE 都包括如下的内容。

（1）PFN，表示虚拟地址对应的物理地址的页号；

（2）Valid，表示对应的页当前是否在物理内存中；

（3）Dirty，表示对应页中的内容是否被修改过；

（4）Use，表示对应页中的内容是否最近被访问过；

（5）AP，访问权限控制，表示操作系统和用户程序对当前这个页的访问权限；

（6）Cacheable，表示对应的页是否允许被缓存。

3.4 加入 TLB 和 Cache

3.4.1 TLB 的设计

1. 概述

到目前为止，对于两级页表（Page Table）的设计，需要访问两次物理内存才可以得到虚拟地址对应的物理地址（一次访问第一级页表，另一次访问第二级页表），而物理内存的运行速度相对于处理器本身来说，有几十倍的差距，因此在处理器执行的时候，每次送出虚拟地址都需要经历上述过程的话，这显然是很慢的（要知道，每次取指令都需要访问存储器）。此时可以借鉴 Cache 的设计理念，使用一个速度比较快的缓存，将页表中最近使用的 PTE 缓存下来，因为它们在以后还可能继续使用，尤其是对于取指令来说，考虑到程序本身的串行性，会顺序地从一个页内取指令，此时将 PTE 缓存起来是大有益处的，能够加快一个页内

4KB内容的地址转换速度。

由于历史的原因，缓存PTE的部件一般不称为Cache，而是称之为TLB(Translation Lookaside Buffer)，在TLB中存储了页表中最近被使用过的PTE，从本质上来将，TLB就是页表的Cache。但是，TLB又不同于一般的Cache，它只有时间相关性(Temporal Locality)，也就是说，现在访问的页，很有可能在以后继续被访问，至于空间相关性(Spatial Locality)，TLB并没有明显的规律，因为在一个页内有很多的情况，都可能使程序跳转到其他不相邻的页中取指令或数据，也就是说，虽然当前在访问一个页，但是未必会访问它相邻的页，正因为如此，Cache设计中很多的优化方法，例如预取(prefetching)，是没有办法应用于TLB中的。

既然TLB本质上是Cache，那么就有以前讲过的三种组织方法，直接相连(direct-mapped)、组相连(set-associative)和全相连(fully-associative)，一般为了减少TLB缺失(miss)发生的频率，会使用全相连的方式来设计TLB，但是这样导致TLB的容量不能太大，因此也有一些设计中采用了组相连的方式来实现容量比较大的TLB。容量过小的TLB会影响处理器的性能，因此在现代的处理器中，很多都采用两级TLB，第一级TLB采用哈佛结构，分为指令TLB(I-TLB)和数据TLB(D-TLB)，一般采用全相连的方式；第二级TLB是指令和数据共用，一般采用组相连的方式，这种设计方法和多级Cache是一样的。

因为TLB是页表的Cache，那么TLB的内容是完全来自于页表的，图3.19表示了一个全相连方式的TLB，从处理器送出的虚拟地址首先送到TLB中进行查找，如果TLB对应的内容是有效的(即valid位是1)，则表示TLB命中，可以直接使用从TLB得到的物理地址来寻址物理内存；如果TLB缺失(即valid位是0)，那么就需要访问物理内存中的页表，此时有如下两种情况。

(1) 在页表中找到的PTE是有效的，即这个虚拟地址所属的页存在于物理内存中，那么就可以直接从页表中得到对应的物理地址，使用它来寻址物理内存从而得到需要的数据，同时将页表中的这个PTE写回到TLB中，供以后使用。

(2) 在页表中找到的PTE是无效的，即这个虚拟地址所属的页不在物理内存中，造成这种现象的原因很多，例如这个页在以前没有被使用过，或者这个页已经被交换到了硬盘中等，此时就应该产生Page Fault类型的异常，通知操作系统来处理这个情况，操作系统需要从硬盘中将相应的页搬移到物理内存中，将它在物理内存中的首地址放到页表内对应的PTE中，并将这个PTE的内容写到TLB中。

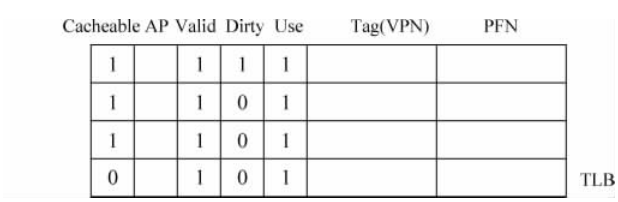

图3.19 TLB的内容

在图3.19中，因为TLB采用了全相连的方式，所以相比页表，多了一个Tag的项，它保存了虚拟地址的VPN，用来对TLB进行匹配查找，TLB中其他的项完全来自于页表，每当发生TLB缺失时，将PTE从页表中搬移到TLB内，TLB中每一项的内容在上面已经介

绍过，此处不再赘述。

在很多处理器中，还支持容量更大的页，因为随着程序越来越大，4KB 大小的页已经不能够满足要求了，对于一个有着 128 个表项(entry)的 I-TLB 来说，只能映射到 128×4KB=512KB 大小的程序，这对于当代的程序来说显然是不够的，因此需要使用容量更大的页，例如 1MB 或是 4MB 大小的页，这样可以使 TLB 映射到更大的范围，避免频繁地对 TLB 进行替换。当然，更大的页也是存在缺点的，对于很多程序来说，如果它利用不到这么大的页，那么就会造成一个页内的很多空间被浪费了，这种现象称为页内的碎片(Page Fragment)，它降低了页的利用效率，而且，每次发生 Page Fault 时，更大的页也就意味着要搬移更多的数据，需要更长的时间才能将这样大的页从下级存储器(如硬盘)搬移到物理内存中，这样使 Page Fault 的处理时间变得更长了。

为了解决这种矛盾，在现代的处理器中都支持大小可变的页，由操作系统进行管理，根据不同应用的特点选用不同的大小的页，这样可以最大限度地利用 TLB 中有限的空间，同时又不至于在页内产生过多的碎片，为了支持这种特性。在 TLB 中需要相应的位进行管理，举例来说，在 MIPS 处理器的 TLB 中，有一个 12 位的 Pagemask 项，它用来指示当前被映射的页的大小，如表 3.3 所示[13]。

表 3.3 MIPS 使用 Pagemask 来指定页的大小

Pagemask	Page Size	Pagemask	Page Size
0000_0000_0000	4KB	0000_1111_1111	1MB
0000_0000_0011	16KB	0011_1111_1111	4MB
0000_0000_1111	64KB	1111_1111_1111	16MB
0000_0011_1111	256KB		

采用不同大小的页，在寻址 TLB 时，进行的地址比较也是不同的，例如，当采用 1MB 大小的页时，只需要将 VA[31:20]作为 Tag，参与地址比较就可以了，虚拟地址剩余的 20 位将用来寻址页的内部；不仅如此，在后文中还会讲到，对 TLB 的寻址还受到其他内容的影响(例如 ASID 和 Global 位)，这些都是在真实的处理器中需要考虑的内容。

在 TLB 以上所有的项中，除了使用位(use)和脏状态位(dirty)之外，其他的项在 TLB 中是不会改变的，它们的属性都是只读，以 D-TLB 为例，当执行 load/store 指令时，都会访问 D-TLB，如果是 TLB 命中，会将 TLB 中对应的使用位(use)置为 1，表示这个页被访问过；如果执行了 store 指令，还会将脏状态位(dirty)也置为 1，表示这个页的某些内容被改变了。因此当 TLB 采用写回(Write Back)的方式，在 TLB 中的某个表项被替换时，也只有这两个位需要被写回到页表中，其他的部分因为在 TLB 中是不会发生变化的，也就没有必要再写回到页表中了。

2. TLB 缺失

当一个虚拟地址查找 TLB，发现需要的内容不在其中时，就发生了 TLB 缺失(miss)，由于 TLB 本身的容量很小，所以 TLB 缺失发生的频率还是比较高的，在很多情况下都可以发生 TLB 缺失，它们主要有如下几种。

(1) 虚拟地址对应的页不在物理内存中，此时在页表中就没有对应的 PTE，由于 TLB 是页表的 Cache，所以 TLB 的内容应该是页表的一个子集，也就是说，在页表中不存在的

PTE，也不可能出现在 TLB 中。

（2）虚拟地址对应的页已经存在于物理内存中了，因此在页表中也存在对应的 PTE，但是这个 PTE 还没有被放到 TLB 中，这种情况是经常发生的，毕竟 TLB 的容量远小于页表。

（3）虚拟地址对应的页已经存在于物理内存中了，因此在页表中也存在对应的 PTE，这个 PTE 也曾经存在于 TLB 中，但是后来从 TLB 中被踢了出来（例如由于这个页长时间没有被使用而被 LRU 替换算法选中），现在这个页又重新被使用了，此时这个 PTE 就存在于页表中，而不在 TLB 内。

其实，第（2）和第（3）两种情况本质上都是在说同一件事情，那就是虚拟地址和物理地址的映射关系存在于页表中，而不存在于 TLB 内，此时只需要从页表中就可以找到这个映射关系，因此它们的处理时间相比于第（1）中情况，肯定是要短的。

解决 TLB 缺失的本质就是要从页表中找到对应的映射关系，并将其写回到 TLB 内，这个过程称为 Page Table Walk，可以使用硬件的状态机来完成这个事情，也可以使用软件来做这个事情，它们各有优缺点，在现代的处理器中均有采用，它们各自的工作过程如下。

（1）软件实现 Page Table Walk，软件实现可以保持最大的灵活性，但是一般也需要硬件的配合，这样可以减少软件工作量，一旦发现 TLB 缺失，硬件把产生 TLB 缺失的虚拟地址保存到一个特殊寄存器中，同时产生一个 TLB 缺失类型的异常，在异常处理程序中，软件使用保存在特殊寄存器当中的虚拟地址去寻址物理内存中的页表，找到对应的 PTE，并写回到 TLB 中，很显然，处理器需要支持直接操作 TLB 的指令，如写 TLB、读 TLB 等。对于超标量处理器来说，由于对异常进行处理时，会将流水线中所有的指令进行抹掉（在后文会介绍这个过程），这样会产生一些性能上的损失，但是使用软件方式，可以实现一些比较灵活的 TLB 替换算法，MIPS 和 Alpha 处理器一般采用这种方法处理 TLB 缺失。但是，为了防止在执行 TLB 缺失的异常处理程序时再次发生 TLB 缺失，一般都将这段程序放到一个不需要进行地址转换的区域（这个异常处理程序一般属于操作系统的一部分，而操作系统就放在不需要地址转换的区域），这样处理器在执行这段异常处理程序时，相当于直接使用物理地址来取指令和数据，避免了再次发生 TLB 缺失的情况。使用软件处理 TLB 缺失的过程可以用图 3.20 来表示。

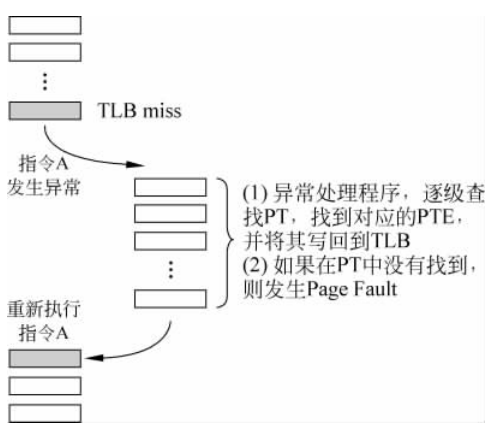

图 3.20　软件处理 TLB miss 的流程

(2) 硬件实现 Page Table Walk，硬件实现一般由内存管理单元(MMU)完成，当发现 TLB 缺失时，MMU 自动使用当前的虚拟地址去寻址物理内存中的页表，前面说过，多级页表的最大优点就是容易使用硬件进行查找，只需要使用一个状态机，逐级进行查找就可以了，如果从页表中找到的 PTE 是有效的，那么就将它写回到 TLB 中，这个过程全部都是由硬件自动完成的，软件不需要做任何事情，也就是这个过程对于软件是完全透明的。当然，如果 MMU 发现查找到的 PTE 是无效的，那么硬件就无能为力了，此时 MMU 会产生 Page Fault 类型的异常，由操作系统来处理这个情况。使用硬件处理 TLB 缺失的这种方法更适合超标量处理器，它不需要打断流水线，因此从理论上来说，性能也会好一些，但是这需要操作系统保证页表已经在物理内存中建立好了，并且操作系统也需要将页表的基地址预先写到处理器内部对应的寄存器中(例如 PTR 寄存器)，这样才能够保证硬件可以正确地寻址页表，ARM、PowerPC 和 x86 处理器都采用了这种方法。

采用软件处理 TLB 缺失可以减少硬件设计的复杂度，尤其是在超标量处理器中，可以采用普通的异常处理过程(在后文会进行介绍)，当这个异常处理完毕后，会重新将这条发生 TLB 缺失的指令取到流水线中，此时这条指令就可以正常执行了。而采用硬件处理 TLB 缺失则会复杂一些，除了需要使用硬件状态机来寻址页表之外，还需要将整个流水线都暂停，等待 MMU 处理这个 TLB 缺失，只有它处理完了，才可以使流水线继续执行。从时间的角度来看，软件处理 TLB 缺失时，除了对应的异常处理程序本身需要占据时间外，还需要考虑到从异常处理程序退出后，将流水线恢复到 TLB 缺失发生之前的状态所需要的时间，这两部分都是由于 TLB 缺失而使处理器耽搁的时间。而反观使用硬件来处理 TLB 缺失的方法，由于只需要将流水线全部暂停，等到硬件处理完毕后，流水线就可以马上从暂停的状态开始继续执行，因此处理器由于 TLB 缺失而耽搁的时间只有硬件处理的那部分时间，相比于软件处理的方式，它不需要将指令重新取回到流水线的过程，相对来看可以省下一些时间，如图 3.21 所示。

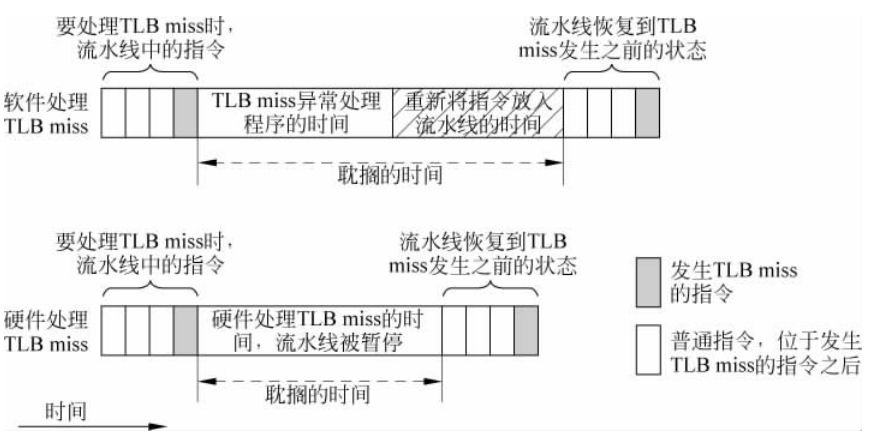

图 3.21 软硬件处理 TLB miss 的对比

MMU 硬件寻址页表所耗费的时间，一般情况下要小于 TLB 缺失的异常处理程序的执行时间，但是两者的差别不会很多。在软件处理方法中，执行 TLB 缺失的异常处理程序之前，需要将流水线清空，等到从异常处理程序退出后，又需要从发生异常的地方(也就是产生

TLB缺失的那条指令)重新开始取指令到流水线中。由于处理器会等到发生TLB缺失的指令到达流水线末尾的时候才进行处理,这需要将流水线中的全部指令都清空。当流水线不深时,如普通的处理器,被清空的指令并不是很多,但是在超标量处理器中,流水线一般比较深,此时被清空的指令就很多了,将这些指令重新取回到流水线中所耗费的时间也会比较长,这也导致在超标量处理器中,软件处理TLB缺失需要更长的时间。

当发生TLB缺失时,如果所需要的PTE在页表中,则TLB缺失的处理时间大约需要十几个周期;如果由于PTE不在页表中而发生Page Fault,则处理时间就需要成百上千个周期了,此时不管采用硬件处理还是软件处理TLB缺失,都不会有明显的差别。TLB缺失发生的频率对于处理器性能的影响是很大的,在典型的页大小为4KB的系统中,只要此时运行的指令或者数据在4KB的边界之内,就不会发生TLB缺失,一般普通的串行程序都会满足这个规律。当然,TLB缺失发生的频率还取决于TLB的大小以及关联度,还有页的大小等因素。一旦由TLB缺失转变成了Page Fault,所需要的处理时间就取决于页的替换算法,以及被替换的页是否是脏状态等因素了。

对于组相连(set-associative)或全相连(fully-associative)结构的TLB,当一个新的PTE被写到TLB中时,如果当前TLB中没有空闲的位置了,那么就要考虑将其中的一个表项(entry)进行替换。理论上说,Cache中使用的替换方法在TLB这里都可以使用,例如最近最少使用算法(Least Recently Used,LRU),但是实际上对于TLB来说,随机替换算法(Random)是一种比较合适的方法,当然,在实际的设计中很难实现严格的随机,此时仍然可以采用一种称为时钟算法(Clock Algorithm)的方法来实现近似的随机。它的工作原理本质上就是一个计数器,这个计数器一直在运转,例如每周期加1,计数器的宽度由TLB中表项的个数来决定,例如一个全相连的TLB中,表项的个数是128个,则计数器的宽度需要7位,当TLB中的内容需要被替换时,就会访问这个计数器,使用计数器当前的值作为被替换表项的编号,这样就近似地实现了一种随机的替换,这种方法从理论上来说,可能并不能获得最优化的结果,但是它不需要复杂的硬件,也能够获得很好的性能,因此综合看起来是一种不错的折中方法。

3. TLB的写入

在前面已经讲过,当一个页从硬盘搬移到物理内存之后,操作系统需要知道这个页中的内容在物理内存中是否被改变过,如果没有被改变过,那么当这个页需要被替换时(例如由于物理内存中的空间不足),可以直接进行覆盖,因为总能从硬盘中找到这个页的备份。而如果这个页的内容在物理内存中曾经被修改过(例如store指令的地址落在了这个页内),那么在硬盘中存储的页就过时了,在物理内存中的这个页要被替换时,不能够直接将其覆盖,而是需要先将它从物理内存中写回到硬盘,因此需要在页表中,对每个被修改的页加以标记,称为脏状态位(dirty),当物理内存中的一个页要被替换时,需要首先检查它在页表中对应PTE的脏状态位,如果它是1,那么就需要先将这个页写回到硬盘中,然后才能将其覆盖。

但是在使用了TLB作为页表的缓存之后,处理器送出的虚拟地址会首先访问TLB,如果命中,那么直接从TLB中就可以得到物理地址,不需要再访问页表。执行load/store指令都会使TLB中对应的“使用位(use)”变为1,表示这个页中的某些数据最近被访问过;如果是store指令,还会使脏状态位(dirty)也变为1,表示这个页中的某些数据被改变了。但

是，如果 TLB 采用写回(Write Back)方式的实现策略，那么使用位(use)和脏状态位(dirty)改变的信息并不会马上从 TLB 中写回到页表，只有等到 TLB 中的一个表项要被替换的时候，才会将它对应的信息写回到页表中，这种工作方式给操作系统进行页替换带来了新的问题，因为此时在页表中记录的状态位(主要是 use 和 dirty 这两位)有可能是“过时”的，操作系统无法根据这些信息，在 Page Fault 发生时找出合适的页进行替换。一种比较容易想到的解决方法就是当操作系统在 Page Fault 发生时，首先将 TLB 中的内容写回到页表，然后就可以根据页表中的信息进行后续的处理了，这个办法显然会耗费一些时间，例如对一个表项个数是 64 的 TLB 来说，需要写 64 次物理内存才能将 TLB 中的内容全部写回到页表，问题是，真的需要这个过程吗？

实际上，这个过程是可以省略的，操作系统完全可以认为，被 TLB 记录的所有页都是需要被使用的，这些页在物理内存中不能够被替换。操作系统可以采用一些办法来记录页表中哪些 PTE 被放到了 TLB 中，而且这样做还有一个好处，它避免了当物理内存中一个页被踢出之后，还需要查找它在 TLB 中是否被记录了，如果是，需要在 TLB 中将其置为无效，因为在页表中已经没有这个映射关系了，因此 TLB 中也不应该有。总结起来就是 TLB 中记录的所有页都不允许从物理内存中被替换。

操作系统在 Page Fault 时，如果从物理内存中选出的要被替换的页是脏(dirty)的状态，那么首先需要将这个页的内容写回到硬盘中，然后才能够将其覆盖。但是，如果在系统中使用了 D-Cache，那么在物理内存中每个页的最新内容都可能存在于 D-Cache 中，要将一个页的内容写回到硬盘，首先需要确认 D-Cache 中是否保留着这个页中的数据。按照前面的说法，TLB 中记录的页都是安全的，操作系统不可能将这些页进行替换，如果在 D-Cache 中保存的数据都属于 TLB 记录的范围，那么操作系统在进行页替换时就不需要理会 D-Cache。但是事实并不总是这样，D-Cache 中的数据未必就一定在 TLB 记录的范围之内，举例来说，当发生 TLB 缺失时，需要从页表中将一个新的 PTE 写到 TLB 中，如果 TLB 此时已经满了，那么就需要替换掉 TLB 中的一个表项，也就不再记录这个页的映射关系，但是这个页的内容在 D-Cache 中仍旧是存在的，当然操作系统此时可以选择同步地将这个页在 D-Cache 中的内容也写回到物理内存中，但是这不是必需的。这样就导致操作系统在物理内存中选择一个页进行替换时，如果这个页是脏(dirty)的状态，那么它最新的内容不一定在物理内存中，而是有可能在 D-Cache 中，此时操作系统就必须能从 D-Cache 中找出这个页的内容，并将其写回到物理内存中，这要求操作系统有控制 D-Cache 的能力，这部分内容将在本章后文进行介绍。

4. 对 TLB 进行控制

当发生 TLB 缺失时，处理器完全可以使用硬件，例如 MMU，自动从页表中找到对应的 PTE 并写回到 TLB 中，只要不发生 Page Fault，整个的过程都是硬件自动完成，不需要软件做任何事情，从这个角度来看，似乎不需要对 TLB 进行什么管理，但是需要注意的是，由于 TLB 是页表的缓存，所以 TLB 中的内容必然是页表的子集，也就是说，如果由于某些原因导致一个页的映射关系在页表中不存在了，那么它在 TLB 中也不应该存在，而操作系统在一些情况下，会把某些页的映射关系从页表中抹掉，例如：

(1) 当一个进程结束时，这个进程的指令(code)、数据(data)和堆栈(stack)所占据的页表就需要变为无效，这样也就释放了这个进程所占据的物理内存空间。但是，此时在 I-TLB

中可能存在这个进程的程序(code)对应的 PTE,在 D-TLB 中可能还存在着这个进程的数据(data)和堆栈(stack)对应的 PTE,此时就需要将 I-TLB 和 D-TLB 中,和这个进程相关的所有内容都置为无效,如果没有使用 ASID(进程的编号,后文会进行介绍),最简单的做法就是将 I-TLB 和 D-TLB 中的全部内容都置为无效,这样保证新的进程可以使用一个干净的 TLB;如果实现了 ASID,那么只将这个进程对应的内容在 TLB 中置为无效就可以了。

(2) 当一个进程占用的物理内存过大时,操作系统可能会将这个进程中一部分不经常使用的页写回到硬盘中,这些页在页表中对应的映射关系也应该置为无效,此时当然也需要将 I-TLB 和 D-TLB 中对应的内容置为无效,但是,一般操作系统会尽量避免将存在于 TLB 中的页置为无效,因为这些页在以后很可能会被继续使用。

因此,抽象出来,对 TLB 的管理需要包括的内容有如下几点。

① 能够将 I-TLB 和 D-TLB 的所有表项(entry)置为无效;

② 能够将 I-TLB 和 D-TLB 中某个 ASID 对应的所有表项(entry)置为无效;

③ 能够将 I-TLB 和 D-TLB 中某个 VPN 对应的表项(entry)置为无效。

不同的处理器有着不同的方法来对 TLB 进行管理,本节以 ARM 和 MIPS 处理器为例进行说明。在 ARM 处理器中,使用系统控制协处理器(ARM 称之为 CP15)中的寄存器对 TLB 进行控制,因此处理器只需要使用访问协处理器的指令(MCR 和 MRC)来向 CP15 中对应的寄存器写入相应的值,就可以对 TLB 进行操作;而在 MIPS 中,则直接提供了对 TLB 进行操作的指令,软件直接使用这些指令对 TLB 进行管理。这两种风格最后都可以得到同样的效果,下面分别对其进行介绍,当然,限于篇幅,本书不会对这两个处理器中相关的所有寄存器都涉及到,关于它们更详细的内容需要参考各自处理器的参考手册。

1) ARM 风格的 TLB 管理[12]

为了实现上面规定的对 TLB 管理的功能,ARM 在协处理器 CP15 中提供了如下的控制寄存器(以 I-TLB 和 D-TLB 分开的架构为例)。

(1) 用来管理 I-TLB 的控制寄存器,主要包括以下几种。

① 将 TLB 中 VPN 匹配的表项(entry)置为无效的控制寄存器,但是 VPN 相等并不是唯一的条件,还需要需要满足下面两个条件。

- 如果 TLB 中一个表项的 Global 位无效,则需要 ASID 也相等;
- 如果 TLB 中一个表项的 Global 位有效,则不需要进行 ASID 比较。

这个控制寄存器如图 3.22 所示。

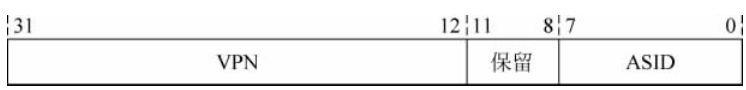

图 3.22　控制 TLB 的寄存器——使用 VPN

当一个进程中某些地址的映射信息被改变时,例如进行了重映射(remap),此时该进程中的这些地址在物理内存中的位置也就改变了,需要将 TLB 中对应的表项也随之置为无效,这就需要使用图 3.22 所示的控制寄存器了。

② 将 TLB 中 ASID 匹配的所有表项(entry)置为无效的控制寄存器,但是 TLB 中那些 Global 位有效的表项不会受到影响,这个控制寄存器如图 3.23 所示。

当一个进程退出时,例如进行进程切换,需要将当前进程在 TLB 中的所有内容都置为

无效，此时就需要使用这个控制寄存器了。

③ 将 TLB 中所有未锁定(unlocked)状态的表项(entry)置为无效，那些锁定(locked)状态的表项则不会受到影响。为了加快处理器中某些关键程序的执行时间，可以 TLB 中的某些表项设为锁定状态，这些内容将不会被替换掉，这样保证了快速的地址转换。

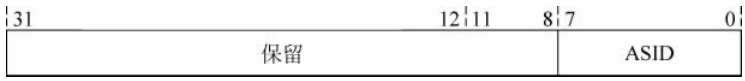

图 3.23 控制 TLB 的寄存器——使用 ASID

(2) 用来管理 D-TLB 的控制寄存器，它们和 I-TLB 控制寄存器的工作原理是一样的，也可以通过 VPN 和 ASID 对 TLB 进行控制，此处不再赘述。

(3) 为了便于对 TLB 中的内容进行控制和观察，还需要能够将 TLB 中的内容进行读出和写入，如图 3.24 所示。

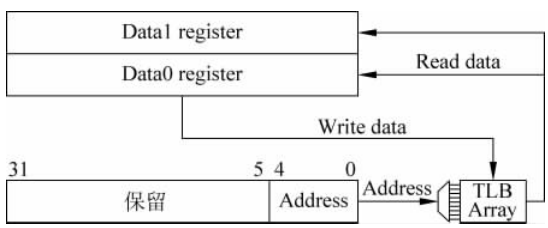

图 3.24 读取 TLB 和写入 TLB

由于 TLB 中一个表项(entry)的内容大于 32 位，所以使用两个寄存器来对应一个表项，如图 3.24 中的 data0 寄存器和 data1 寄存器，在 ARM 的 Cortex A8 处理器中，这两个寄存器位于协处理器 CP15 中。当读取 TLB 时，被读取表项的内容会放到这两个寄存器中；而在写 TLB 时，这两个寄存器中的内容会写到 TLB 中，当然，要完成这个过程，还需要给出寻址 TLB 的地址，例如一个表项个数是 32 的 TLB，需要 5 位的地址来寻址，这个地址放在指令中指定的一个通用寄存器中。一般在调试处理器的时候，需要使用图 3.24 所示的功能。

总结来说，在 ARM 处理器中对 TLB 的控制是通过协处理器来实现的，因此只需要使用访问协处理器的指令(MCR 和 MRC)就可以了，其实不仅是对于 TLB，在 ARM 处理器中对于存储器的管理，例如 Cache 和 BTB 等部件，都是通过协处理器来实现的，这种方式虽然比较灵活，但是却不容易使用，在没有用户手册的情况下，很难记住协处理器中每个寄存器的用处，不过考虑到 ARM 采用了硬件方式来解决 TLB 缺失，一般情况下并不需要直接使用指令对 TLB 进行操作，因此这种协处理器的控制风格也是无可厚非的。

2) MIPS 风格的 TLB 管理[14]

在 ARM 处理器中，发生 TLB 缺失时，查找页表并将 PTE 写回到 TLB 的过程使用硬件来实现，但是在 MIPS 的架构定义中，TLB 缺失的解决过程是由软件完成的，为了支持这样的操作，MIPS 中定义了专门操作 TLB 的指令，使用这些指令可以直接对 TLB 进行控制，这些指令如表 3.4 所示(假设 TLB 中表项的个数是 64 个，全相连结构，因此需要 6 位的地址进行寻址)。

表 3.4 MIPS 中对 TLB 进行控制的指令

指令	描 述	需要使用的寄存器
TLBP	Probe TLB for matching entry，使用虚拟地址 VA 来查找 TLB 中是否存在它的映射关系，如果存在，将对应 entry 的地址放到 Index 寄存器中	EntryHi：用来存储寻址 TLB 需要使用的 VPN，这个寄存器也包括了 ASID 部分 Index：将 TLB 中被寻址到的 entry 的地址放到这个寄存器中，如果在 TLB 中找到对应的映射关系，则 Index 寄存器的[32]置为 0，[5:0]存储地址；如果没有在 TLB 中找到，则将 Index 寄存器的[32]置为 1
TLBR	Read indexed TLB entry，从 TLB 中将 Index 寄存器指定 entry 的内容读出来，放到寄存器 EntryHi 和 EntryLo 中	Index：用来存储寻址 TLB 的地址，会使用这个寄存器的[5:0]来寻址 TLB EntryHi：被 Index 寄存器寻址到的 TLB entry 中的 VPN 部分会放到这个寄存器中 EntryLo：被 Index 寄存器寻址到的 TLB entry 中的其他内容会被放到这个寄存器中
TLBWI	Write Indexed TLB entry，向 Index 寄存器寻址到的 TLB entry 中写入 EntryHi 和 EntryLo 的内容	Index：用来存储寻址 TLB 的地址，使用[5:0]来寻址 TLB EntryHi：用来存储写入到 TLB entry 中的 VPN EntryLo：用来存储写入到 TLB entry 的其他内容
TLBWR	Write Random TLB entry，使用 Random 寄存器来寻址 TLB，向被寻址的 entry 中写入 EntryHi 和 EntryLo 的内容	Random：一个用来产生随机值的寄存器，只有[5:0]有效，用来寻址 TLB，多使用一个计数器来模拟随机的过程 EntryHi：用来存储写入到 TLB entry 中的 VPN EntryLo：用来存储写入到 TLB entry 的其他内容

为了实现对 TLB 的完全控制，在 MIPS 中设计了四条指令 TLBP、TLBR、TLBWI 和 TLBWR，以及四个寄存器 Index、EntryHi、EntryLo 和 Random，使用这四条指令和四个寄存器，可以实现对 TLB 的控制，尤其是在 TLB 缺失对应的异常处理程序中，TLBWR 这条指令使用的频率最高，它可以实现 TLB 中随机的替换策略，一个典型的 TLB 缺失的异常处理程序如图 3.25 所示。。

```
mfc0   $k1, context   //将寻址PT的地址放到寄存器$k1中
lw     $k1, 0($k1)    //寻址PT，将得到的PTE放到寄存器$k1中
mtc0   $k1, EntryLo   //将PTE放到寄存器EntryLo中
tlbwr                 //将EntryLo和EntryHi寄存器的内容随机写到TLB中
eret                  //从TLB异常处理程序中退出
```

图 3.25 MIPS 中，一个典型的 TLB miss 的异常处理程序

前文说过，要寻址页表，需要两部分内容，即页表的基地址（例如 PTR 寄存器）和页表内的偏移（例如虚拟地址 VA[31:12]）。在 MIPS 处理器中，为了加快寻址页表的过程，硬件会自动将这两部分内容放到一个寄存器中，它就是上面程序中的 context 寄存器，它位于协处理器 CP0 中，软件可以直接使用 context 寄存器中的内容作为寻址页表的地址。当然，由于 load 指令无法直接使用 CP0 中的寄存器，所以首先要把 context 寄存器放到通用寄存器 $k1 中，注意在 MIPS 架构中约定，通用寄存器中的 R26 和 R27 只用在中断和异常的处理

程序中，它们被称为 $k0 和 $k1。TLBWR 指令将 EntryHi 和 EntryLo 两个寄存器的内容写到 TLB 内随机指定的一个表项中，不过在发生 TLB 缺失的异常时，硬件会自动将当前未能转换的虚拟地址的 VPN，以及当前进程的 ASID，写到 EntryHi 寄存器中，不需要软件再去组织 EntryHi 寄存器的内容，因此软件只需要组织 EntryLo 寄存器的内容即可。其实，这个寄存器存储的就是页表中被寻址到的 PTE 的内容。当然，如果要使用 TLBP 指令，那么仍旧需要软件来组织 EntryHi 寄存器的内容。

3.4.2 Cache 的设计

1. Virtual Cache

TLB 只是加速了从虚拟地址到物理地址的转换，可以很快地得到所需要的数据（或指令）在物理内存中的位置，也就是得到了物理地址，但是，如果直接从物理内存中取数据（或指令），显然也是很慢的，因此可以使用在以前章节提到的 Cache 来缓存物理地址到数据的转换过程。实际上，从虚拟地址转化为物理地址之后，后续的过程就和前文讲述的内容是一样的了。因为这种 Cache 使用物理地址进行寻址，因此称为物理 Cache（Physical Cache），使用 TLB 和物理 Cache 一起进行工作的过程如图 3.26 所示。

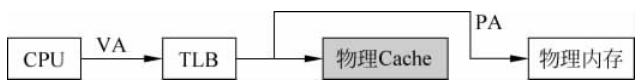

图 3.26 Physical Cache

很显然，如果不使用虚拟存储器，处理器送出的地址会直接访问物理 Cache，而现在需要先经过 TLB 才能再访问物理 Cache，因此必然会增加流水线的延迟，如果还想获得和以前一样的运行频率，就需要将访问 TLB 的过程单独使用一级流水线，但是这样就增加了分支预测失败时的惩罚（penaly），也增加了 load 指令的延迟，不是一个很好的做法。

既然图 3.26 所示的过程，最终要从虚拟地址得到对应的数据，那么为什么不使用 Cache 来直接缓存从虚拟地址到数据的关系呢？

当然是可以的，因为这个 Cache 使用虚拟地址来寻址，称之为虚拟 Cache（Virtual Cache），用来和前面的物理 Cache 相对应。既然使用虚拟 Cache，可以直接从虚拟地址得到对应的数据，那么是不是就可以不使用 TLB 了呢？当然不是这样，因为一旦虚拟 Cache 发生了缺失，仍旧需要将对应的虚拟地址转换为物理地址，然后再去物理内存中获得对应的数据，因此还需要 TLB 来加速从虚拟地址到物理地址的转换过程。在使用虚拟 Cache 的这种方法中，如果能够从中得到数据，那么是最好的情况，否则仍旧需要经过 TLB 并访问物理内存，这个过程如图 3.27 所示。

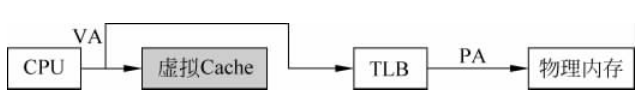

图 3.27 Virtual Cache

在图 3.27 中，如果使用虚拟地址，从虚拟 Cache 中找到了需要的数据，那么就不需要再访问 TLB 和物理内存，在流水线中使用这种虚拟 Cache，不会对处理器的时钟周期产生明显的负面影响，不过它会引入一些新的问题，需要耗费额外的硬件进行解决，这是由于虚拟

地址的属性和物理地址是不同的，每个物理地址总是有且只有一个物理内存中的位置和它对应，两个不同的物理地址必然对应物理内存中两个不同的位置，因此，在前文中使用物理Cache时，不会有任何的问题，但是，直接使用虚拟Cache则会引入新的问题，主要可以概括为两方面。

同义问题(synonyms)，也称做重名(aliasing)，即多个不同的名字对应相同的物理位置，在虚拟存储器的系统中，一个进程内或者不同的进程之间，不同的虚拟地址可以对应同一个物理内存中的位置，例如前文中提到的位于操作系统中的printf函数，许多进程都可以使用它。如果使用了虚拟Cache，由于直接使用虚拟地址进行寻址，则不同的虚拟地址会占用Cache中不同的地方，那么当很多虚拟地址都对应着一个物理地址时，就会导致在虚拟Cache中，这些虚拟地址虽然占据了不同的地方，但是它们实际上就是对应着同一个物理内存中的位置，这样会引起两方面的问题，一是浪费了宝贵的Cache空间，造成了Cache等效容量的减少，降低了整体的性能；二是当执行一条store指令而写数据到虚拟Cache中时，只会将这个虚拟地址在Cache中对应的内容进行修改，而实际上，Cache中其他有着相同物理地址的地方都需要被修改(这些位置的虚拟地址不同，但是都对应着同一个物理地址)，否则，其他的虚拟地址读取Cache时，就无法得到刚才更新过的正确值了，上面描述的问题如图3.28所示。

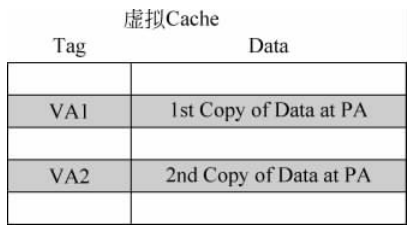

图3.28　存在重名问题的Virtual Cache

在图3.28中，当执行一条store指令而向虚拟地址VA1写入数据时，虚拟Cache中VA1对应的内容会被修改，但是在Cache中，还存在虚拟地址VA2也映射到同一个物理地址PA，这样一个物理地址在虚拟Cache中有两份数据，当一条load指令使用虚拟地址VA2读取虚拟Cache时，就会得到过时的数据了。

并不是所有的虚拟Cache都会发生同义问题，这取决于页的大小和Cache的大小，而前面说过，虚拟地址转化为物理地址的时候，页内的偏移是不变的，也就是说，对于大小为4KB的页来说，虚拟地址的低12位不会发生变化，如果此时有一个直接相连(direct-mapped)结构的虚拟Cache，而这个Cache的容量小于4KB，那么寻址Cache的地址就不会大于12位，此时即使两个不同的虚拟地址对应同一个物理位置，它们寻址虚拟Cache的地址也是相同的，因此会占用虚拟Cache中的同一个地方，不会存在不同的虚拟地址占用Cache中不同位置的情况。相反，只有Cache的容量大于4KB时，才会导致寻址Cache的地址大于12位，此时映射到同一个物理位置的两个不同的虚拟地址，寻址虚拟Cache使用的地址也是不同的。它们会占用Cache内不同的地方，这样就会出现同义问题，如图3.29所示为在大小为8KB、直接相连结构的Cache中出现同义问题的示意图。

由于在虚拟地址到物理地址的转换过程中，只有页内偏移保持不变，所以在图3.29中，

当两个虚拟地址映射到同一个物理地址时，两个虚拟地址的第 12 位可能是 0、也可能是 1。也就是说，此时在虚拟 Cache 中会有两个不同的地方存储着同一个物理地址的值，此时最简单的方法就是当一个虚拟地址写 Cache 时，将 Cache 中可能出现同义问题的两个位置都进行更新，这就相当于将它们作为一个位置来看待，要实现这样的功能，就需要使用物理地址作为 Cache 的 Tag 部分，并且能够同时将虚拟 Cache 中两个可能重名的位置都读取出来，这就要在 Cache 中使用 bank 的结构，例如图 3.29 所示的大小为 8KB、直接相连结构的 Cache 就需要两个 bank，使用虚拟地址 VA[11:0]进行寻址。在写 Cache 时，两个 bank 中对应的位置都需要更新，这样在读取 Cache 时，也就会从两个 bank 中得到同一个值了，但是这样的方法相当于将 Cache 的容量减少了一半，显然无法在实际当中使用。其实，利用这种 bank 结构的 Cache，可以在不减少 Cache 容量的前提下解决同义问题，这种方法如图 3.30 所示。

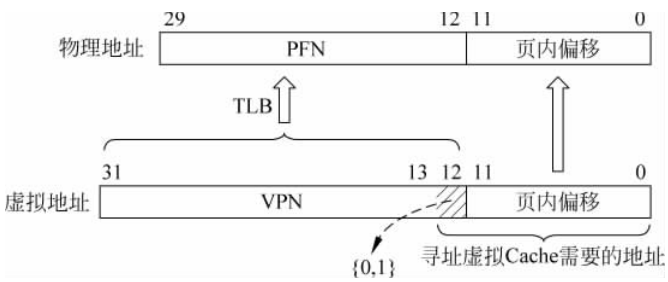

图 3.29　一个物理地址可以有两个虚拟地址与之对应

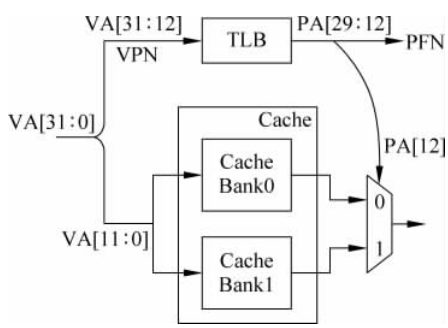

图 3.30　使用 bank 的方法解决重名问题

在图 3.30 中，一个大小为 8KB、直接相连结构的 Cache，内部被分为了两个 4KB 的 bank，使用 VA[11:0]作为两个 bank 公用的地址，需要注意的是，在 Cache 中的数据部分和 Tag 部分都使用了图 3.30 所示的 bank 结构，这种 Cache 的读写过程如下。

① 读取 Cache 时，两个 bank 都会被 VA[11:0]寻址而得到对应的内容，这两个 bank 的输出被送到一个由物理地址 PA[12]控制的多路选择器上，当物理地址 PA[12]为 0 时，选择 bank0 输出的值，当物理地址 PA[12]为 1 时，选择 bank1 输出的值。由于物理地址需要经过 TLB 才可以得到，所以当 Cache 中两个 bank 输出的值送到多路选择器的时候，物理地址可能还没有从 TLB 中得到，这样在一定程度上增加了处理器的周期时间。需要注意的是，由于 Cache 中的 Tag 部分也采用了这种 bank 结构，所以使用图 3.30 所示的方法可以

得到 Cache 最终输出的 Tag 值，这个值会和 TLB 输出的 PFN 值进行比较，从而判断 Cache 是否命中。

② 写 Cache 的时候，由于一条指令只有退休(retire)的时候，才会真正地写 Cache，此时的物理地址已经得到了，所以在写 Cache 时可以根据物理地址 PA[12]的值，将数据写到对应的 bank 中。

这种方法相当于将 PFN 为偶数(PA[12]为 0)的所有地址写到了 Cache 的 bank0 中，将 PFN 为奇数(PA[12]为 1)的所有地址写到了 bank1 中，这样不会造成 Cache 存储空间的浪费。当然缺点是增加了一些硬件复杂度，并且在 Cache 中 way 的个数不增加的前提下，随着 Cache 容量的增大，需要使用的硬件也越来越多，例如使用大小为 16KB、直接相连结构的 Cache 时，就需要使用四个 bank，采用 PA[13:12]来控制多路选择器。由于采用了 bank 结构，Cache 的输入需要送到所有的 bank，造成输入端的负载变得更大，而且每次读取 Cache 时所有的 bank 都会参与动作，所以功耗相比普通的 Cache 也会增大，这些都是采用 bank 结构的虚拟 Cache 需要面临的缺点。

(2) 同名问题(homonyms)，即相同的名字对应不同的物理位置，在虚拟存储器中，因为每个进程都可以占用整个虚拟存储器的空间，因此不同的进程之间会存在很多相同的虚拟地址。而实际上，这些虚拟地址经过每个进程的页表转化后，会对应不同的物理地址，这就产生了同名问题：很多相同的虚拟地址对应着不同的物理地址。当从一个进程切换到另一个进程时，新的进程使用虚拟地址来访问虚拟 Cache 的话(假设使用虚拟地址作为 Cache 中的 Tag 部分)，从 Cache 中可能得到上一个进程的虚拟地址对应的数据，这样就产生了错误。为了避免这种情况发生，最简单的方法就是在进程切换的时候将虚拟 Cache 中所有的内容都置为无效，这样就保证了一个进程在开始执行的时候，使用的虚拟 Cache 是干净的。同理，对于 TLB 也是一样的，在进程切换之后，新的进程在使用虚拟地址访问 TLB 时，可能会得到上一个进程中虚拟地址的映射关系，这样显然也会产生错误，因此在进程切换的时候，也需要将 TLB 的内容清空，保证新的进程使用的 TLB 是干净的。当进程切换很频繁时，就需要经常将 TLB 和虚拟 Cache 的内容清空，这样可能浪费了大量有用的值，降低了处理器的执行效率。

既然无法直接从虚拟地址中判断它属于哪个进程，那么就为每个进程赋一个编号，每个进程中产生的虚拟地址都附上这个编号，这个编号就相当于是虚拟地址的一部分，这样不同进程的虚拟地址就肯定是不一样了。这个编号称为 PID(Process ID)，当然更通用的叫法是 ASID(Address Space IDentifier)，使用 ASID 相当于扩展了虚拟存储器的空间，此时仍然是每个进程可以看到整个的 4GB 的虚拟存储器空间，而且每个进程的 4GB 都互相不交叠，如图 3.31 所示。

例如，当使用 8 位的 ASID 时，那么虚拟存储器就有 $2^8=256$ 个 4GB 的空间，就相当于此时虚拟存储器的空间为 $256\times4GB=1024GB$，也就是说，使用 ASID 就等于扩大了虚拟存储器的空间。

但是，使用 ASID 也引入了一个新的问题，当多个进程想要共享同一个页时，如何实现这个功能呢？这就需要在 ASID 之外再增加一个标志位，称之为 Global 位，或简称为 G 位。当一个页不只是属于某一个进程，而是被所有的进程共享时，就可以将这个 Global 位置为 1，这样在查找页表的时候，如果发现 G 位是 1，那么就不需要再理会 ASID 的值，这样就实

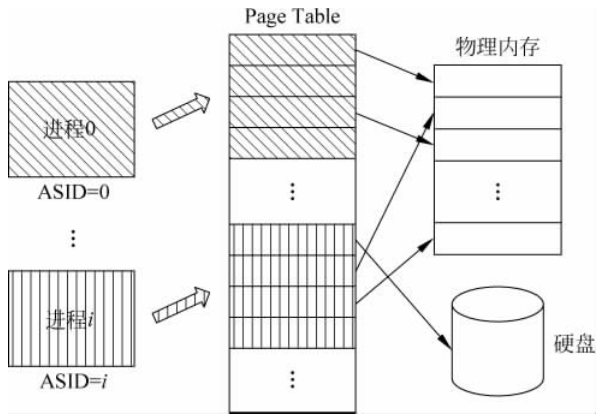

图 3.31 使用 ASID 之后的地址转换

现了一个页被所有进程共享的功能。

ASID 和原来的虚拟地址一起组成了新的虚拟地址，这就相当于使虚拟地址的位数增加了，例如在 32 位的处理器中采用 8 位的 ASID，则相当于虚拟地址是 40 位，此时还可以使用两级的页表，将这 40 位的虚拟地址进行划分，假设仍旧使用大小为 4KB 的页，查找第一级页表使用 14 位，查找第二级页表使用 14 位，那么此时第一级页表和第二级页表的大小都是 $2^{14} \times 4B = 64KB$，过大的页表可能导致其内部出现碎片，降低页表的利用效率。

为了解决这个问题，可以采取三级页表的方式。增加一个额外的页表，它使用 ASID 进行寻址，这个页表中的每个 PTE 都存放着第二级页表的基地址，如图 3.32 所示，此时仍旧需要使用 PTR 寄存器存放第一级页表的基地址。

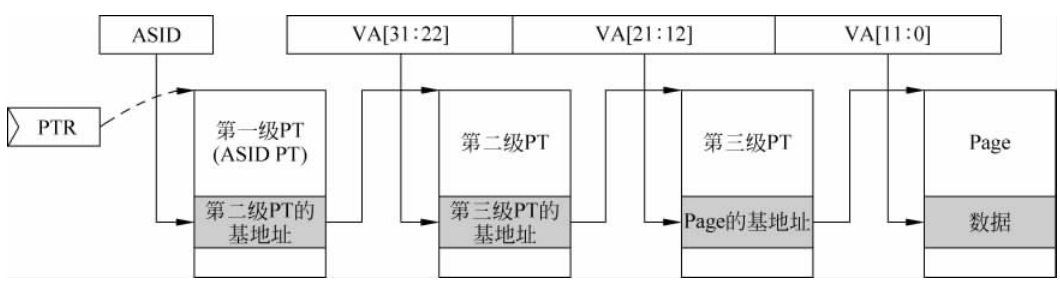

图 3.32 使用多级页表来解决 ASID 加入之后引起的问题

当然，使用这样的方式，要从虚拟地址中得到需要的数据，相比于两级页表，需要多一次物理内存的访问，这会造成 TLB 缺失的处理时间变长，是使用 ASID 带来的一个负面影响，尤其是 TLB 缺失发生的频率很高时，这种负面影响更为严重。

在使用多级页表的系统中，只有第一级页表才会常驻物理内存中，第一级页表的基地址由处理器当中专用的寄存器指定，例如图 3.32 中的 PTR 寄存器。支持 ASID 的处理器中还会有一个寄存器来保存当前进程的 ASID 值，每次操作系统创建一个进程时，就会给当前的进程分配 ASID 值，并将其写到 ASID 寄存器中，这个进程中所有的虚拟地址都会在前面被附上这个 ASID 的值，在 TLB 中，ASID 和 VPN 一起组成了新的虚拟地址，参与地址的比较，这样就在 TLB 中对不同的进程进行了区分。

当系统中运行的进程个数超过 ASID 能够表示的最大范围时，例如有多于 256 个进程存在于 8 位 ASID 的系统中，此时就需要操作系统从已经存在的 256 个 ASID 中挑出一个不经常使用的值，将它在 TLB 中对应的内容清空，并将这个 ASID 分配给新的进程。由于此时新的进程会更新 PTR 寄存器，为了能够对旧的进程进行恢复，操作系统需要将被覆盖的 PTR 寄存器的值保存起来，这样等到这个旧的进程再次被执行时，就可以知道它存在于物理内存的哪个位置了。总结起来就是说，如果使用了 ASID，那么操作系统就需要管理 ASID 的使用，尤其是对于那些已经退出的进程，要及时地回收它的 ASID 值，以供新的进程使用。

2. 对 Cache 进行控制

采用哈佛结构的处理器中，Cache 分别被指令（I-Cache）和数据（D-Cache）使用，在 Cache 中缓存着物理内存的内容，因此 Cache 中的内容都是物理内存的子集。但是，要保持这种关系，需要对下列情况进行特殊处理。

（1）当 DMA 需要将物理内存中的数据搬移到其他地方，但此时物理内存中最新的数据还存在于 D-Cache 中，因此在进行 DMA 搬移之前，需要将 D-Cache 中所有脏状态（dirty）的内容写回到物理内存中。

（2）当 DMA 从外界搬移数据到物理内存的一个地址上，而这个地址又在 D-Cache 中被缓存，那么此时就需要将 D-Cache 中这个地址的内容置为无效。

（3）当发生 Page Fault 时，需要从硬盘中读取一个页并写到物理内存中，如果物理内存中被覆盖的页是脏的状态，并且这个页内的部分内容还存在于 D-Cache 中，那么就需要先将 D-Cache 中的内容写回到物理内存，然后才能将这个页写回到硬盘中，此时就可以对物理内存中的这个页进行覆盖了。

（4）处理器有可能执行一些“自修改”的指令，将处理器后续要执行的一些指令进行修改，这些新的指令会先作为数据写到 D-Cache 中，如果想要处理器能够正确地执行这些被修改过的指令，需要将 D-Cache 中的这些内容都写回到物理内存中，同时需要将 I-Cache 中的所有内容都清空，这样才能够保证处理器能够执行到最新的指令。在处理器中，D-Cache 和 I-Cache 之间没有直接的通路，只能通过物理内存进行交互。

将上面的所有操作进行抽象，得出需要对 Cache 进行的操作有如下几种。

（1）能够将 I-Cache 内的所有 Cache Line 都置为无效；

（2）能够将 I-Cache 内的某个 Cache Line 置为无效；

（3）能够将 D-Cache 内的所有 Cache Line 进行 clean；

（4）能够将 D-Cache 内的某个 Cache Line 进行 clean；

（5）能够将 D-Cache 内的所有 Cache Line 进行 clean，并置为无效；

（6）能够将 D-Cache 内的某个 Cache Line 进行 clean，并置为无效。

在上面的描述中，Cache 的 clean 操作指的是将脏状态（dirty）的 Cache line 写回到物理内存的过程。

和 TLB 多采用全相连的结构不同，Cache 一般采用组相连的结构，为了能够找到 Cache 中的某个 Cache line，可以使用的寻址手段有以下两种。

（1）Set/Way，通过提供 set 和 way 的信息，可以定位到 Cache 中的一个 Cache line。

（2）地址，在 Cache 真正工作的时候，是通过地址来查找 Cache 的。这个地址可以是虚拟地址，也可以是物理地址，这取决于采用物理 Cache 还是虚拟 Cache。这个地址的 Index

部分用来找到 Cache 中的某个 Cache set，Tag 部分用来从不同的 way 中选出匹配的 Cache line(如果 Cache 中使用物理地址作为 Tag，还需要先将虚拟地址转化为物理地址)。

和 TLB 的管理类似，Cache 的管理也有不同的风格，比较典型的两个例子就是 ARM 处理器和 MIPS 处理器，在 ARM 处理器中，仍旧使用系统协处理器(即 CP15)中的寄存器来管理 Cache，通过管理 CP15 中对应的寄存器，就可以实现对 Cache 的控制；而在 MIPS 处理器中，则直接使用专用的指令来管理 Cache。从易用性方面来看，MIPS 处理器使用的方法无疑要占据优势，更容易被程序员使用，不过从执行效率来看，两者并没有明显的区别，都在现实世界中被广泛地使用。

1) ARM 风格的 Cache 管理[12]

在 ARM 处理器中，对 Cache 的管理主要依靠系统协处理器 CP15 中的寄存器来实现，通过访问协处理器的指令(MRC 和 MCR)，可以将指定的值送到对应功能的协处理器寄存器中，这些寄存器主要用来对 I-Cache 和 D-Cache 进行控制，它们主要包括下面的内容，如表 3.5 所示。

表 3.5 ARM 中对 Cache 进行管理的协处理器

	CRn	Op1	CRm	Op2	Data	功 能
操作 I-Cache	c7	0	c5	0	0	将 I-Cache 全部置为无效
			c5	1	VA	根据地址将某个 I-Cache line 置为无效
操作 D-Cache			c6	0	0	将 D-Cache 全部置为无效
			c6	1	VA	根据地址将某个 D-Cache line 置为无效
			c6	2	Set/way	根据 set/way 将某个 D-Cache line 置为无效
			c10	1	VA	根据地址将某个 D-Cache line 进行 clean
			c10	2	Set/way	根据 set/way 将某个 D-Cache line 进行 clean
			c14	1	VA	根据地址将某个 D-Cache line 进行 clean，并置为无效
			c14	2	Set/way	根据 set/way 将某个 D-Cache line 进行 clean，并置为无效

在表 3.5 所示中，CRn、Op1、CRm、Op2 和 Data 都是指令 MRC 和 MCR 中携带的内容，MRC 和 MCR 指令的格式如下。

```
MRC{<cond>}<coproc>, <Op1>, <Rt>, <CRn>, <CRm>{, <Op2>}
MCR{<cond>}<coproc>, <Op1>, <Rt>, <CRn>, <CRm>{, <Op2>}
```

MRC 指令将协处理器中某个寄存器的内容放到处理器内部指定的通用寄存器中，MCR 指令将处理器内部某个通用寄存器的内容放到协处理器内部指定的寄存器中，这两条指令都可以条件执行，在 ARM 的体系结构中定义了 CP0～CP15 共 16 个协处理器，但是最常用的就是 CP15 协处理器，在其中定义了各种控制寄存器，用来对处理器的工作状态进行控制，从本质上来说，Op1、CRn、CRm 和 Op2 共同决定了协处理器中的某个寄存器，而 Rt 指定了处理器内部的某个通用寄存器，其中存放的内容即是表 3.5 中的 Data 部分。举例来说，MCR 指令的用法如图 3.33 所示。

不管使用 VA 还是 set/way 信息来查找 Cache line，都需要将 VA 和 set/way 的信息放到通用寄存器 Rt 中，在 ARM 中定义了它们的格式。使用这种方式，可以对 Cache 完成指

定的操作，例如可以使用 set/way 的信息对整个 D-Cache 进行 clean 操作，只需要使用一段程序，逐步地增加 Rt 寄存器中 set/way 的值，从而遍历到整个 D-Cache 就可以了。在前文说过，发生 Page Fault 并需要将某个页的内容进行替换时，就会使用到这个功能，通过控制 set/way 从而将整个 D-Cache 进行 clean 操作，可以保证将最新的内容更新回到物理内存中，从而保证程序的正确执行。

```
MCR  p15, 0, <Rt>, c7, c5, 0 ;  将I-Cache的内容全部置为无效，Rt寄存器中存放的内容应该是0
MCR  p15, 0, <Rt>, c7, c6, 1 ;  根据地址VA，将D-Cache的某个line置为无效，Rt寄存器中存放的内容应该是VA
MCR  p15, 0, <Rt>, c7, c6, 2 ;  根据set/way，将D-Cache的某个line置为无效，Rt寄存器中存放的内容应该是set/way
```

图 3.33　使用 MCR 指令对 Cache 进行控制

对于 I-Cache 来说，程序是无法直接向其中写入内容的，所以对于它来说，不存在数据是否脏(dirty)的问题，也就不需要进行 clean 操作。但是在某些情况下，程序可能会有自修改(self-modifying)的功能，也就是说，程序自己改变后面要执行的内容，修改后面的某些指令，在使用 Cache 的系统中，要实现这样的功能，是没有办法通过 I-Cache 来实现的，而是必须借助于 D-Cache，将新的指令作为普通的数据，使用 store 类型的指令，将其写到 D-Cache 中，然后将 D-Cache 进行 clean 操作，使这些被修改的指令可以真正更新到 L2 Cache 中(假设 L2 Cache 已经可以被 I-Cache 和 D-Cache 共享)，然后再将整个 I-Cache 中的内容置为无效，这样就可以保证程序在继续执行时，可以取到最新的指令来执行，如果没有将 D-Cache 进行 clean 操作，那么程序仍然有可能从 I-Cache 中执行到旧的指令，从而使自修改的功能无法得到正确的执行。

当然，在 ARM 处理器中，对 Cache 的控制还有很多其他的功能，例如可以将某个 Cache line 进行锁定，用来保证这个 Cache line 不会被替换；可以读取/写入 Cache 中某个指定的 Cache line；可以控制 I-Cache 和 D-Cache 是否被使用(事实上，处理器刚上电的时候都不会开启 Cache，而是从一段不可映射、不可 Cache 的地址区域开始执行指令，等到将 Cache 初始化之后才会进行开启)等，具体的功能参见 ARM 处理器的架构参考手册。

2) MIPS 风格的 Cache 管理[14]

在 MIPS 处理器中，直接使用指令来完成对 Cache 的控制，这条指令就是 CACHE 指令，它的格式如图 3.34 所示。

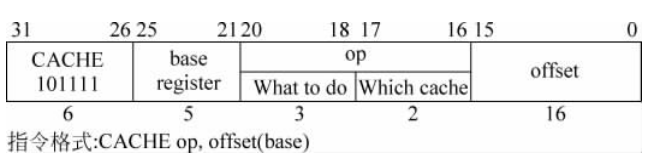

图 3.34　MIPS 中的 CACHE 指令

这条指令和普通 load/store 指令的格式是类似的，都是由指令中指定的寄存器加上 16 位的立即数来共同产生一个地址，这个地址被 MIPS 称为 EA(Effective Address)，即 EA＝GPR[base]＋offset。这个地址可以用作普通的虚拟地址从而直接寻址 Cache，当然在

MIPS 处理器中，也可以使用 set/way 的信息来直接寻址 Cache，set/way 的信息在这个地址 EA 中直接指定，总结起来，这个有效地址 EA 的用处如表 3.6 所示。

表 3.6 MIPS 中，CACHE 指令所携带 EA 的用处

EA 的用处	Cache 的类型	解　　释
地址	虚拟 Cache	EA 被用作虚拟地址，对 Cache 进行寻址
地址	物理 Cache	EA 被用作物理地址，对 Cache 进行寻址
Set/way	虚拟 Cache/ 物理 Cache	EA 中包含了 set/way 的信息，使用它直接对 Cache 进行寻址，此时 EA 的内容如下所示： 31　　0 未使用 | Way | Set | Byte-within-line

在 MIPS 处理器中对 Cache 的管理完全通过 CACHE 指令来实现，指令中的 op 部分（位于指令[20:16]）指定了操作的类型，这明显不同于 ARM 中使用协处理器中的寄存器对 Cache 进行控制的方法，从易用性方面来说，MIPS 的这种做法更容易被接受。在 5 位的 op 部分中，后 2 位（即指令[17:16]）用来指定对何种 Cache 进行操作，它的内容如下所示。

```
2'b00 = L1 I - Cache
2'b01 = L1 D - Cache
2'b10 = L3 Cache,如果存在的话
2'b11 = L2 Cache,如果存在的话
```

指令中 op 部分的高 3 位（即指令[20:18]）用来指定操作的类型，前面提到的对 Cache 控制的功能都可以在这部分找到，如表 3.7 所示。

表 3.7 MIPS 对 Cache 的控制

指令[20:18]	命　　令	EA 的功能	解　　释
3'b000	Index Write back invalidate	Set/way	将指定的 Cache line 置为无效，对于 D-Cache 来说，如果被找到的 line 是 dirty 状态，那么首先应该将其写回到下级存储器中（即 clean 操作），然后才能置为无效
3'b001	Index Load Tag	Set/way	将指定的 Cache line 中的 Tag 部分放到 CP0 的 TagLo 和 TagHi 寄存器中，将 line 中被寻址到的两个字放到 CP0 的 DataLo 和 DataHi 寄存器中
3'b010	Index Store Tag		将 CP0 中的 TagLo 和 TagHi 寄存器的内容写到指定 Cache line 的 Tag 部分，它可以用来初始化 Cache
3'b011	保留		
3'b100	Hit invalidate	地址	将指定的 Cache line 置为无效，而不管这个 line 是否是 dirty 状态，当这个功能应用于 D-Cache 时，可能会导致部分数据丢失，一般都用来对 I-Cache 进行操作
3'b101	Hit Write back invalidate	地址	对指定的 Cache line 进行 clean 操作，然后将这个 line 置为无效
3'b110	Hit Write back	地址	对指定的 Cache line 进行 clean 操作，当 DMA 或者其他 CPU 要读取物理内存中的数据时，需要首先将 D-Cache 中的数据更新回到物理内存中

续表

指令[20:18]	命 令	EA 的功能	解 释
3'b111	Fetch and Lock	地址	从指定的 Cache line 中读取想要的数据，如果这个数据不在 Cache 中，那么就从下级存储器中将其取出来并写到这个 line 中，如果这个 line 是 dirty 状态，还需要首先将它的数据写回到下级存储器中。从 Cache line 中得到需要的数据后，就将这个 line 置为锁定的状态，当以后发生 Cache miss 时，这个 line 将不会被替换掉 要想解除这个 line 的锁定状态，需要使用 CACHE 指令将这个 line 置为无效，这可以通过命令 Index Writeback invalidate、Hit invalidate 或 Hit Write back invalidate 来实现

由表 3.7 可以看出，在 MIPS 处理器中使用 CACHE 指令，可以完成对 Cache 的控制，相比于 ARM 使用协处理器中的寄存器进行控制的方式，MIPS 的这种做法更容易被程序员使用，这也符合 MIPS 一贯简洁的风格。

3.4.3 将 TLB 和 Cache 放入流水线

1. Physically-Indexed, Physically-Tagged

在使用虚拟存储器的系统中，仍旧可以使用物理 Cache，这是最保守的一种做法，因为处理器送出的虚拟地址(VA)会首先被 TLB 转换为对应的物理地址(PA)，然后使用物理地址来寻址 Cache，此时就像是没有使用虚拟存储器一样，直接使用了物理 Cache，并且使用物理地址的一部分作为 Tag，因此将其称为 physically-indexed，physically-tagged Cache，这种设计的示意图如图 3.35 所示。

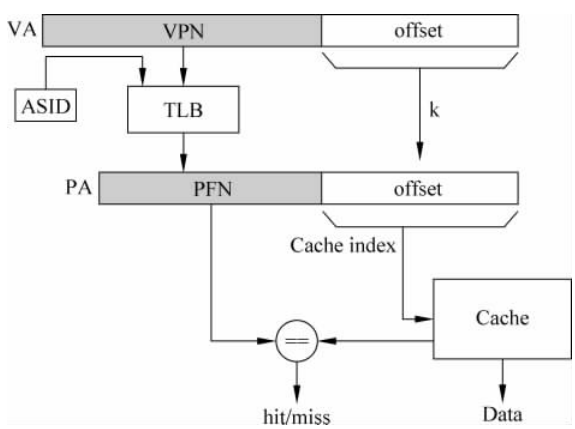

图 3.35 physically-indexed，physically-tagged Cache

由于寻址 TLB 的过程也需要消耗一定的时间，为了不至于对处理器的周期时间造成太大的负面影响，可能需要将访问 TLB 的过程单独作为一个流水段，这样相当于增加了流水线的级数，对于 I-Cache 来说，增加一级流水线会导致分支预测失败时有更大的惩罚(penalty)；而对于 D-Cache 来说，增加一级流水线会造成 load 指令的延迟(latency)变大。

由于 load 指令一般都是在相关性的顶端，因此这种方法会对其他相关指令的唤醒(wake-up)造成一定的负面影响。这种设计方法在理论上是完全没有问题的，但是在真实的处理器中很少被采用，因为它完全串行了 TLB 和 Cache 的访问。而实际上，这是没有必要的，因为从虚拟地址到物理地址的转换过程中，低位的 offset 部分是保持不变的，在图 3.35 所示的设计中，如果物理地址中寻址 Cache 的部分使用 offset 就足够的话，那么就不需要等到从 TLB 中得到物理地址之后才去寻址 Cache，而是直接可以使用虚拟地址的 offset 部分，这样访问 TLB 和访问 Cache 的过程是同时进行的，如图 3.36 所示。

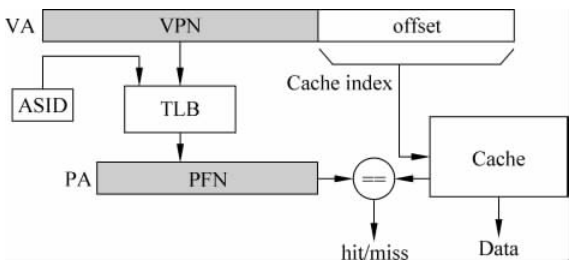

图 3.36　同时访问 Cache 和 TLB

这种设计在不影响流水线深度的情况下获得很好的性能，但是对 Cache 的大小有了限制，实际上，图 3.36 是 virtually-indexed，physically-tagged Cache 的一种情况，下面会对其进行介绍。

2. Virtually-Indexed，Physically-Tagged

这种方式使用了虚拟 Cache，根据 Cache 的大小，直接使用虚拟地址的一部分来寻址这个 Cache(这就是 virtually-indexed)，而在 Cache 中的 Tag 则使用物理地址中的 PFN(这就是 physically-tagged)，这种 Cache 是目前被使用最多的，大多数的现代处理器都使用了这种方式，如图 3.37 所示为在直接映射(direct-mapped)结构的 Cache 中使用这种方式的设计。

在这样的方法中，访问 Cache 和访问 TLB 是可以同时进行的，假设在直接映射的 Cache 中，每个 Cache line 中包括 2^b 字节的数据，而 Cache set 的个数是 2^L，也就是说，寻址 Cache 需要的地址长度是 $L+b$，它直接来自于虚拟地址，再假设页的大小是 2^k 字节，因此就有了图 3.37 中的三种情况。

(1) $k > L + b$，此时 Cache 的容量小于一个页的大小；

(2) $k = L + b$，此时 Cache 的容量等于一个页的大小；

(3) $k < L + b$，此时 Cache 的容量大于一个页的大小。

从虚拟地址到物理地址的转换过程中，offset 部分(由图 3.37 中的 k 给出)是保持不变的，而在虚拟存储器中最小的单位就是页，图 3.37 中的前两种情况，也就是“$k > L + b$”和“$k = L + b$”实际对应着一种情况，寻址 Cache 的地址此时虽然直接来自于虚拟地址，但是这个地址并没有超过 offset 部分，所以寻址 Cache 的地址在虚拟地址到物理地址的转换过程中是不会发生变化的，这个地址也可以认为是来自于物理地址，只不过它并行访问了 TLB 和 Cache，提高了处理器的执行效率。总结来看，图 3.37 所示本质上对应着两种情况，需要相应地采取不同的设计方法。

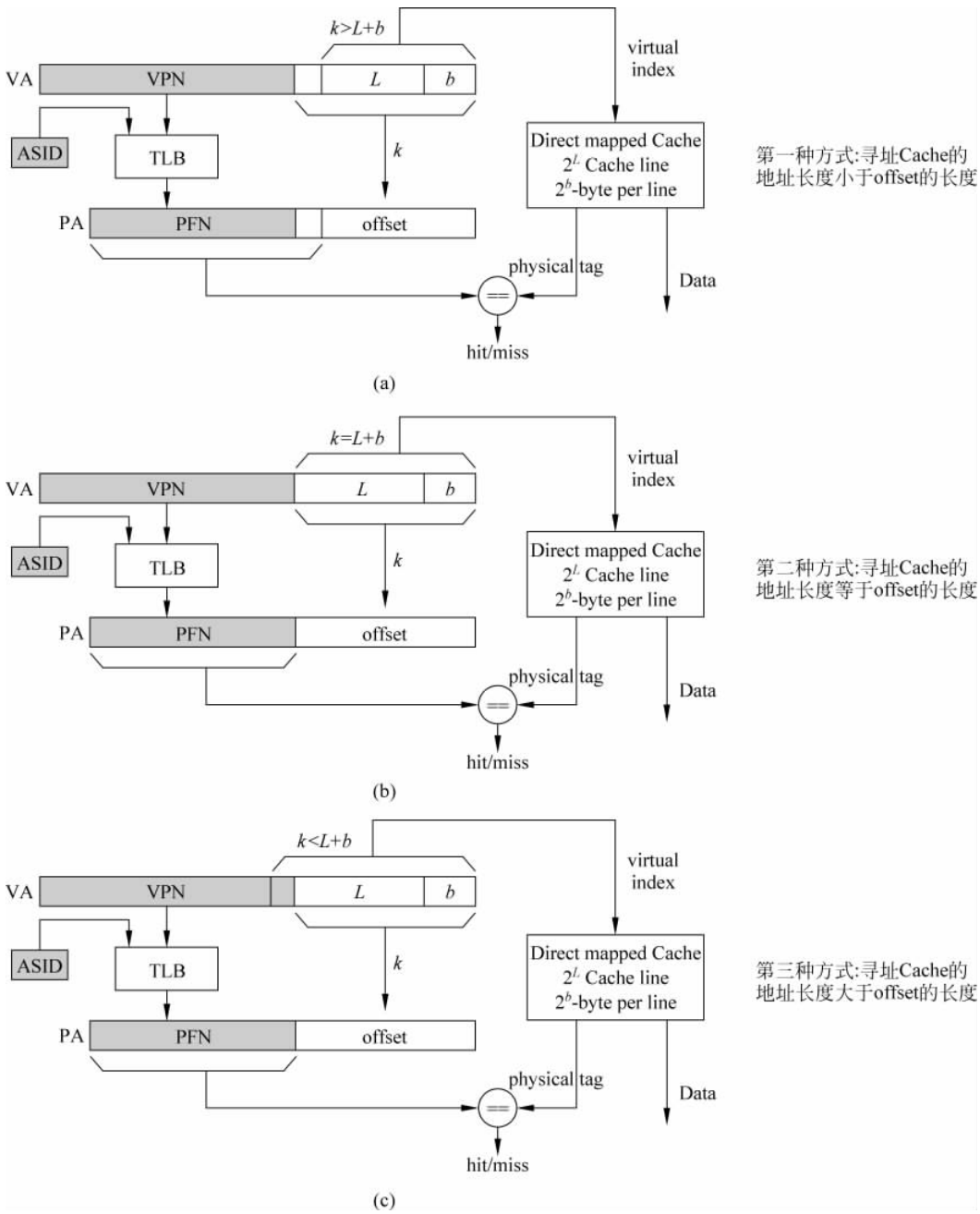

图 3.37　virtually-indexed，physically-tagged 的三种情况

(1) 情况 1：$k = L + b$ 或 $k > L + b$

此时 Cache 容量小于或等于一个页的大小，直接使用虚拟地址中的[$L+b-1:0$]部分来寻址 Cache，找到对应 Cache line 中的数据，并将这个 Cache line 的 Tag 部分和 TLB 转换得到的 PFN 进行比较，用来判断 Cache 是否命中。这种设计可以避免虚拟 Cache 中的重名问题，因为它本质上使用物理地址来寻址 Cache，相当于直接使用了物理 Cache。为什么会

有这样的效果呢？假设有几个不同的虚拟地址映射到同一个物理地址，那么这些虚拟地址的 offset 部分肯定是相同的，也就是这些虚拟地址用来寻址 Cache 的地址是一样的（虚拟地址[$L+b-1:0$]来自于 offset），因此这些不同的虚拟地址必然会位于直接映射（direct-mapped）Cache 中的同一个位置，这样出现重名的这些虚拟地址就不可能同时共存于 Cache 中了，因为 Cache 内只有一个位置可以容纳这些重名的虚拟地址。

还需要注意的是，即使在组相连（set-associative）结构的 Cache 中，这些重名的虚拟地址也不能够占据不同的 way，因为此时本质上使用的是物理 Cache，在物理 Cache 中，同一个物理地址只能占据一个 way，只有 index 部分相同的不同物理地址才可以占据不同的 way。由于重名的这些虚拟地址对应同一个物理地址，所以这些虚拟地址也就不可能占据 Cache 中不同的地方了。

在物理地址中除了寻址 Cache 的 index 部分，剩下的内容就作为 Tag 存储在 Cache 中，这部分内容必须要经过 TLB 才可以得到。访问 TLB 和 Cache 是同时进行的，当从 TLB 得到物理地址的时候，从 Cache 中也读取到了对应的 Tag 值，此时就可以进行 Tag 比较从而判断是否命中，如果处理器的周期时间允许的话，这些过程可以在一个周期内完成。

上面的这种方法使用了直接映射结构的 Cache，它的容量小于等于一个页的大小，因此对于页大小为 4KB 的典型设计来说，Cache 的容量就被限制在了 4KB，不能够再大了。要想使用更大的 Cache，就需要使用组相连结构的 Cache，因为增加 way 的个数不会引起寻址 Cache 所需地址位数的增加，这种方法如图 3.38 所示，举例来说，如果需要一个大小为 32KB 的 Cache，那么就需要 8-way 的设计，Intel 就是这样做的。那如果需要使用一个 4MB 的 Cache，就需要 way 的个数是 1024，这显然很难在现实当中实现，因此使用这种方式，对于 Cache 容量的大小是有限制的。

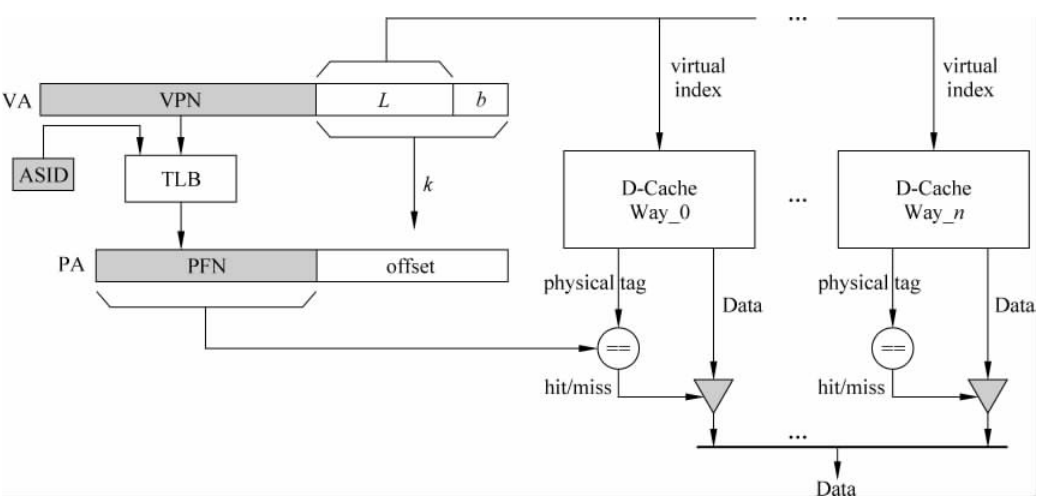

图 3.38　使用更多的 way 数来增加 Cache 的容量

（2）情况 2：$k < L + b$

在上面的情况下想要使用容量比较大的 Cache，考虑到速度的限制，不可能在组相连结构的 Cache 中无限制地增加 way 的个数，因此只能够增加每个 way 的容量，此时就会导致虚拟地址中寻址 Cache 的 index 位数的增加，这就变成了 $L+b$ 的值已经大于 offset 的位数

k 的情况，这种设计就是真正的 virtually-indexed，虽然这样可以在不增加 way 个数的情况下获得容量比较大的 Cache，但是会面临重名的问题：不同的虚拟地址会对应同一个物理地址。举例来说，在页大小为 4KB 的系统中，有两个不同虚拟地址 VA1 和 VA2，映射到同一个物理地址 PA，有一个直接映射结构的 Cache，容量为 8KB，需要 13 位的 index 才可以对其进行寻址，如图 3.39 所示。

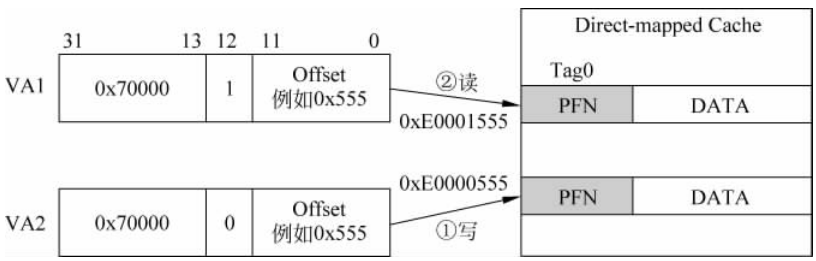

图 3.39　重名问题的一个例子

此时不能保证 VA1 和 VA2 寻址 Cache 的 index 是相同的，因为它们的位[12]有可能不相同，如图 3.39 所示，此时的位[12]已经属于 VPN 了，正因为如此，才造成 VA1 和 VA2 寻址 Cache 的 index 不同，它们会被放到不同的 Cache set 上，这时候问题就出现了：Cache 中两个不同的位置对应着物理存储器中同一个物理位置，这样不但造成了 Cache 空间的浪费，而且当 Cache 中 VA2 对应数据被改变时（例如执行 store 指令向 VA2 中写入值），VA1 的数据不会随着变化，因此就造成了一个物理地址在 Cache 中有两个不同的值，当后续的 load 指令从地址 VA1 读取数据时，就不会从 Cache 中读取到正确的值了。

如何解决这个问题呢？在前面讲述虚拟 Cache 时，介绍了使用 bank 结构的 Cache 来解决这种重名的问题，将所有重名的虚拟地址都分门别类地放到 Cache 中指定的地方，当然方法不止这一种，还可以让这些重名的虚拟地址只有一个存在于 Cache 中，其他的都不允许在 Cache 中存在，这样也可以避免上述的问题，如何实现这种方法呢？

可以使用 L2 Cache 来实现这个功能，使 L2 Cache 中包括所有 L1 Cache 的内容，也就是采用之前介绍的 inclusive 的方式，如图 3.40 所示。

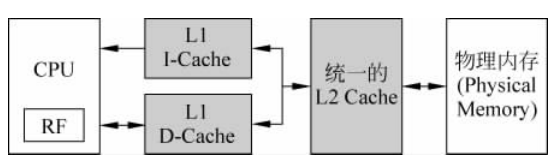

图 3.40　采用统一的 L2 Cache

对第一级的 I-Cache 和 D-Cache 使用一个统一的 L2 Cache，L1 Cache 中的内容是 L2 Cache 的子集。当 L1 Cache 发生缺失时，如果数据（或指令）也不在 L2 Cache 中，则会去物理内存中寻找这个数据（或指令），并将其放到 L1 Cache 的同时，也会放到 L2 Cache 中；当 L1 Cache 中的脏状态（dirty）数据被替换掉之前，会被首先写到 L2 Cache 中。其实，只要保证从物理内存中读取的数据（或指令）在写到 L1 Cache 的同时也写到 L2 Cache 中，就能够保证 L2 Cache 包括所有 L1 Cache 的内容了，当然必须保证 L2 Cache 的容量大于 I-Cache 和 D-Cache 之和。

利用 L2 Cache 解决上述重名的问题的方法如图 3.41 所示。

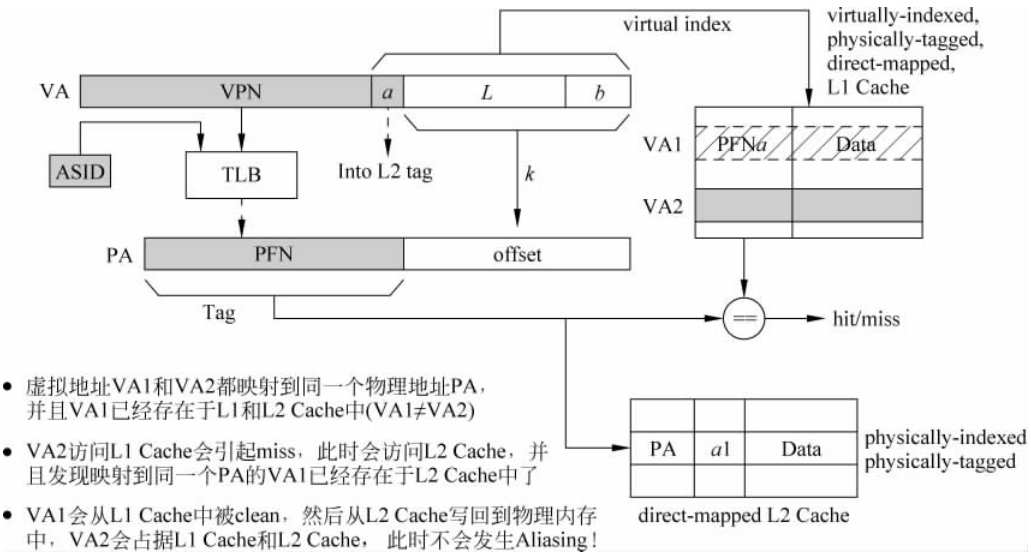

图 3.41　使用 L2 Cache 解决重名问题

假设最开始时，虚拟地址 VA1 已经存在于 L1 Cache 和 L2 Cache 当中，当虚拟地址 VA2 访问 L1 Cache 时，会发生缺失，于是就去 L2 Cache 中寻找数据。在大多数处理器中，L2 Cache 都是纯粹的物理 Cache，也就是采用 physically-indexed，physically-tagged 的方式，此时会直接使用经过 TLB 转化的物理地址(PA)来寻址 L2 Cache，而这个物理地址已经存在于 L2 Cache 中了(因为 VA1 存在于 L2 Cache 中)，所以此时 VA2 发现它要找的数据存在于 L2 Cache 当中，但是此时还应该告诉 VA2，在 L2 Cache 中，PA 的数据其实是属于 VA1 的，因此需要在 L2 Cache 的每个 Cache line 中加以标记。如图 3.41 所示的那样，在 L2 Cache 的每个 Cache line 中存储虚拟地址的 a 部分，这样当 VA2 被转换为 PA 并访问 L2 Cache 时，如果发现这个 PA 已经存在于 L2 Cache 中，并且这个 PA 对应的 a 不等于 VA2 中的 a，那么就表明此时存在和 VA2 重名的虚拟地址(图 3.41 中即是 VA1)，在将 VA2 的数据从 L2 Cache 读取到 L1 Cache 之前，应该使用 L2 Cache 中这个 PA 对应的 a 来寻址 L1 Cache，找到 L1 Cache 中那个存放重名虚拟地址的 Cache line，将其置为无效。当然，如果发现此时这个 Cache line 是脏(dirty)的状态，那么还需要首先将它进行 clean 操作，也就是将这个脏的 Cache line 写回到 L2 Cache 中，然后才能将它置为无效，这样在 L1 Cache 中就不会存在重名的虚拟地址，此时就可以将 VA2 的数据从 L2 Cache 读取到 L1 Cache 中，而且能够保证在 L1 Cache 中不存在和 VA2 重名的情况。

如何使用 L2 Cache 中存储的 a 部分来寻址 L1 Cache 呢？其实是使用 a 和 offset 共同组成的地址来寻址 L1 Cache 的，这从图 3.41 就可以看出来。前面说过，重名的虚拟地址由于都对应同一个物理地址，所以这些虚拟地址的 offset 部分其实都是一样的。在图 3.41 中，VA2 从 L2 Cache 中发现了 VA1 所占据的 Cache line，那么就使用这个 Cache line 中存储的 a，和 VA2 的 offset 组成一个新的地址，也就是{a, offset}，这个地址其实就是 VA1 用来寻址 Cache 的地址部分(再次强调，VA1 和 VA2 是重名的，所以它们寻址 L1 Cache 的地址除了 a 部分不同之外，其他的部分都是一样的)，这样就可以从 L1 Cache 中找到 VA1 所

占据的 Cache line，并将其进行 clean 和置为无效的操作。从这个过程可以看出，只要在 L2 Cache 中存储了虚拟地址中的 a 部分，就可以通过它来消除 L1 Cache 内的重名问题了。

3. Virtually-Indexed, Virtually-Tagged

在这种方式中，直接缓存了从虚拟地址到数据的过程，它会使用虚拟地址来寻址 Cache（也就是 virtually-indexed），并使用虚拟地址作为 Tag（也就是 virtually-tagged），因此这种 Cache 可以称得上是名副其实的 Virtual Cache 了。如果 Cache 命中，那么直接就可以从 Cache 中获得数据，都不需要访问 TLB；如果 Cache 缺失，那么就仍旧需要使用 TLB 来将虚拟地址转换为物理地址，然后使用物理地址去寻址 L2 Cache，从而得到缺失的数据（在现代的处理器中，L2 及其更下层的 Cache 都是物理 Cache），这个过程如图 3.42 所示。

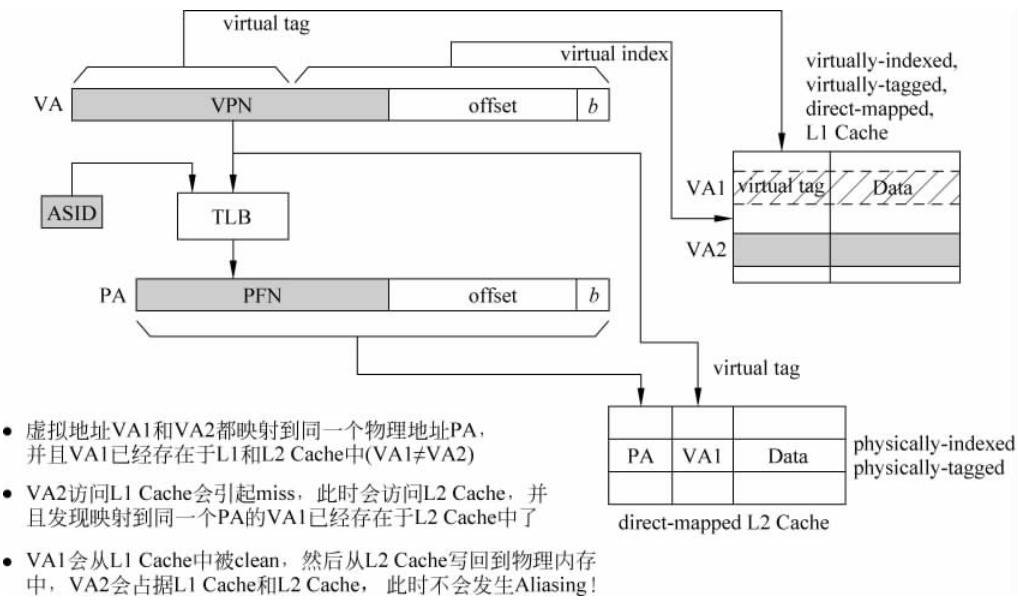

图 3.42　在 Virtual Cache 中，仍旧使用 L2 Cache 解决重名问题

当然，使用这种方法，仍旧会遇到重名的问题，当存在多个虚拟地址对应同一个物理地址的情况时，L1 Cache 中也可能出现一个物理地址占据多个 Cache line 的问题，解决它的方法还可以和上一节一样，让那些重名的虚拟地址只有一个可以存在于 L1 Cache 中，只是此时不能在 L2 Cache 中只存储虚拟地址的 a 部分了，而是应该存储整个的虚拟地址。回想上一节的内容，其实只需要存储虚拟地址的 a 部分就可以在 L1 Cache 中分辨出不同的虚拟地址了，而在本节的设计中，只有将整个虚拟地址都存储在 L2 Cache 中才可以分辨出 L1 Cache 中不同的虚拟地址，因为 L1 Cache 是 virtually-tagged 的结构，会使用虚拟地址的一部分作为 Tag。举例来说，当虚拟地址 VA1 已经存在于 L1 Cache 和 L2 Cache 之中时，和 VA1 重名的 VA2 访问 L1 Cache 时，会引起缺失，此时需要将 VA2 经过 TLB 转化为物理地址（PA），然后访问 L2 Cache，此时发现这个物理地址已经存在于 L2 Cache 当中，而且对应的 Cache line 中存储着 VA1，这就表示一个和 VA2 重名的 VA1 已经存在于 L1 Cache 中。因此，在将 VA2 对应的数据放到 L1 Cache 之前，首先需要将 L1 Cache 中 VA1 对应的那个 Cache line 置为无效，当然，如果它是脏（dirty）的状态，那么首先需要进行 clean 操作，

通过使用这种方法，就能够保证所有重名的虚拟地址只有一个存在于 L1 Cache 中。

4. 小结

到目前为止可以知道，对于访问存储器的指令来说（主要是 load/store 指令），它在执行的时候，涉及到如下四种部件的访问（暂不考虑硬盘）。

- TLB；
- 物理内存中的页表（Page Table）；
- D-Cache；
- 物理内存中的页（Page）。

每种部件都可能是命中或者缺失的，因此总共有 2×2×2×2＝16 种可能的组合，但是考虑到它们之间有如下的关系。

（1）如果 TLB 命中，则物理内存中的页表也必然是命中的；

（2）如果 D-Cache 命中，则物理内存中的页也必然是命中的；

（3）如果物理内存中的页表是命中的，则物理内存中的页也必然是命中的。

经过将这些不可能的组合进行删除，则 load/store 指令在执行的时候可能引起的情况如表 3.8 所示。

表 3.8 访问存储器时各种可能的情况

情况	TLB	Page Table	D-Cache	Page	注 释
当 D-Cache hit 时	假设 hit	由于 TLB hit，则它必然 hit	Hit	由于 D-Cache hit，则它必然 hit	不需要检查物理内存
当 D-Cache miss 时	假设 hit	由于 TLB hit，则它必然 hit	Miss	Hit	D-Cache 被更新
当 TLB miss 时	Miss	Hit	假设 hit	由于 D-Cache hit，则它必然 hit	TLB 被更新，并重新访问 TLB
当 TLB ＋ D-Cache miss 时	Miss	Hit	Miss	Hit	TLB 和 D-Cache 都会被更新，并重新访问 TLB
当 Page Fault 时	必然 miss	必然 miss	必然 miss	必然 miss	进行 Page Fault 的处理

通过表 3.8 所示可以看出，最好的情况发生在 TLB 和 Cache 同时命中的时候，此时 load/store 指令在执行的时候需要经历的流水线最短，并不需要访问物理内存，这种情况也是最希望看到的。如果 TLB 或 Cache 中的某个发生了缺失，则都需要访问物理内存，如果能够在物理内存中找到需要的内容，那么会耽误十几个周期，发生缺失的 load/store 指令会重新被执行。最坏的情况发生在 TLB 和 Cache 都缺失的时候，而且，还伴随着 Page Fault（也就是在物理内存中找不到需要的数据），这时候就需要通知操作系统来处理这个情况了，操作系统会从物理内存的下一级存储器中（一般是硬盘或者闪存）将这个缺失的页搬移到物理内存中，设置好 Page Table 对应的 PTE，并将 TLB 和 Cache 也进行更新，然后才可以重新执行这条发生 Page Fault 的 load/store 指令。访问最下层存储器，例如硬盘，这个过程是以 ms 为单位的，可以想象，对于以 ns 为单位的处理器来说，这个过程会是多么的漫长。为

了提高整体的执行效率，操作系统此时会保存当前进程的状态，转而执行其他的进程，等到从硬盘中读取到了需要的页，并将 Page Fault 处理完毕后，才会继续执行这个被“雪藏”的进程。

总体来看，使用虚拟存储器之后，会给整个处理器的设计带来很多额外的麻烦，但是现代的操作系统对虚拟存储器有着先天的渴求，所以在处理器中必须要支持这种特性，这中间需要很多的折中(trade off)，才可以在芯片的成本、功耗和性能方面得到一个比较合适的结果。