方法：

1. 故障避免技术：通过合理构建系统来避免故障的出现。
2. 故障容忍技术：使用冗余技术，即使出现故障，仍然可以按照需求服务。
3. 故障预测技术：预测故障的出现和构建，从而允许在器件故障前进行替换。

5.5.2　纠正 1 位错、检测 2 位错的汉明编码

理查德·汉明（Richard Hamming）发明了一种广泛应用于存储器的冗余技术，并因此获得 1968 年的图灵奖。二进制数间的距离对于理解冗余码很有帮助。汉明距离是两个等长二进制数对应位置不同的位的数量。例如，011011 和 001111 的距离为 2。如果在一种编码中，码字之间的最小距离为 2，且其中有 1 位错误，将会发生什么？这会将一个有效的码字转化为无效码字。因此，如果能够检测一个码字是否有效，就可以检测出 1 位的错误，称为 1 位**错误检测编码**。

错误检测编码：这种编码方式能够检测出数据中有 1 位错误，但是不能对错误位置进行精确定位，因此不能纠正错误。

汉明使用奇偶校验码进行错误检测。在奇偶校验码中，要计数一个字中 1 的个数是奇数还是偶数。当一个字被写入存储器时，奇偶校验位也被写入（1 表示奇数，0 表示偶数）。也就是说，$N+1$ 位字中 1 的个数永远是偶数。当读出该字时，奇偶校验位也一并读出并检查。如果计算出的校验码与存储的不匹配，则发生错误。

例题

计算值为 $31_{(10)}$、宽度为一字节的二进制数的奇偶性，并写出保存到存储器中的内容。假定奇偶校验位在最右侧，并且最高有效位在存储器中发生了翻转。然后将其读出。能否检测出错误？如果最高两位都发生翻转呢？

答案　31_{10} 是 00011111_2，它有 5 个 1。为使其校验为偶数个 1，需要在奇偶校验位写入 1，即 000111111_2。如果最高有效位发生翻转，将读出 100111111_2，其中有 7 个 1，因为预期有偶数个 1，而计算出有奇数个 1，则报告发生了错误。如果最高两位发生翻转，将读出 110111111_2，其中有 8 个 1，因此无法检测出错误。

如果有 2 位同时出错，则 1 位奇偶校验位技术无法检测到错误，因为码字奇偶性不变。（实际上，1 位奇偶校验可以检测任意奇数个错误，但是实际情况中，发生 3 位错误的概率远低于 2 位错误的概率，所以实际中 1 位奇偶校验码仅用于检测 1 位错误。）

当然，奇偶校验码不能纠正错误，汉明想要做到检错的同时又能纠错。如果码组中最小距离为 3，那么任意发生 1 位错误的码字与其对应的正确码字的距离，要小于它与其他有效码字的距离。他想出了一个容易理解的将数据映射到距离 3 的码字，为纪念汉明，我们将这种方法称为汉明纠错码（ECC）。我们使用额外的奇偶校验位确定单个错误的位置。以下是计算汉明纠错码的步骤：

1. 从左到右由 1 开始依次编号，这与传统的从最右侧由 0 开始编号相反。
2. 将编号为 2 的整数幂的位标记为奇偶校验位（1，2，4，8，16，…）。
3. 剩余其他位用于数据位（3，5，6，7，9，10，11，12，13，14，15，…）。
4. 奇偶校验位的位置决定了其对应的数据位（图 5-24 以图形方式进行了说明），如下所示：

- 校验位 1（0001_2）检查 1，3，5，7，9，11，…位，这些位的编号最右一位为 1（0001_2、

0011_2，0101_2，0111_2，1001_2，1011_2，…）。

- 校验位 2（0010_2）检查 2，3，6，7，10，11，14，15，…位，这些位的编号最右第二位为 1。
- 校验位 4（0100_2）检查 4 ～ 7，12 ～ 15，20 ～ 23，…位，这些位的编号最右第三位为 1。
- 校验位 8（1000_2）检查 8 ～ 15，24 ～ 31，40 ～ 47，…位，这些位的编号最右第四位为 1。

请注意，每个数据位都被至少两个奇偶校验位覆盖。

5. 设置奇偶校验位，为各组进行偶校验。

位编号	1	2	3	4	5	6	7	8	9	10	11	12
编码后的数据位		p_1	p_2	d_1	p_4	d_2	d_3	d_4	p_8	d_5	d_6	d_7	d_8
校验位覆盖范围	p_1	X		X		X		X		X		X	
	p_2		X	X			X	X			X	X	
	p_4				X	X	X	X					X
	p_8								X	X	X	X	X

图 5-24　8 位数据的汉明纠错码，包括奇偶校验位、数据位及其覆盖范围

如同变魔术一样，你可以通过查看奇偶校验位来确定数据位是否出错。在图 5-24 中的 12 位编码中，如果四个奇偶校验位（p_8，p_4，p_2，p_1）的值为 0000_2，则说明没有错误。但是，如果四个校验位值为 1010_2，即 10_{10}，那么汉明纠错码告诉我们第 10 位（d_6）是错误的。由于数字是二进制的，可以通过翻转第 10 位的值来纠错。

例题

假设一个 8 位数据的值是 10011010_2。首先写出对应的汉明纠错码，然后将第 10 位取反，说明汉明纠错码如何发现并纠正该错误。

为奇偶校验位留出空位，12 位码字为 __1_001_1010。

答案 位置 1 检查 1、3、5、7、9 和 11 位，即 __**1**_**0**0**1**_**1**0**1**0。为使该组的校验性为偶，将第 1 位设置为 0。

位置 2 检查 2、3、6、7、10、11 位，即 0_**1**_0**0****1**_1**0****1**0。为使该组的校验性为奇，将第 2 位设置为 1。

位置 4 检查 4、5、6、7、12 位，即 011_**001**_101，将第 4 位设置为 1。

位置 8 检查 8、9、10、11、12 位，即 0111001_**1010**，将第 8 位设置为 0。

最终的码字是 011100101010。将数据位第 10 位取反后为 011100101110。

奇偶校验位 1 是 0（**011100101110** 有 4 个 1，为偶性；故该组无错误）。

奇偶校验位 2 是 1（**0**1**1**10**0**10**1**1**10** 有 5 个 1，为奇性；故该组某个位置上有错误）。

奇偶校验位 4 是 1（**011100**101**110** 有 2 个 1，为偶性；故该组无错误）。

奇偶校验位 8 是 1（011100**01110** 有 3 个 1，为奇性；故该组某个位置上有错误）。

奇偶校验位 2 和 8 正确。因为 2 + 8 = 10，第 10 位肯定是错误的。因此，我们将其反转为 011100101010，即完成了纠错。

汉明并没有止步于 1 位纠错码。通过再增加 1 位的方式，可以使码组中的最小汉明距离变为 4。这意味着我们可以纠正 1 位错并检测 2 位错。该方法添加了 1 位奇偶校验位，对整个字进行计算校验。例如，对于一个 4 位的字，需要 7 位完成 1 位错误检测。计算汉明奇偶

校验位 H（p_1 p_2 p_3）（依然采用偶校验）最后计算整个字的偶校验位 p_4：

1	2	3	4	5	6	7	8
p_1	p_2	d_1	p_3	d_2	d_3	d_4	$\mathbf{p}_4$

上述纠正 1 位错并检测 2 位错的算法就是像以前一样先计算 ECC 组的奇偶校验位（H），再计算全组的奇偶校验位（p_4）即可。以下是可能出现的 4 种情况：

1. H 为偶并且 p_4 为偶，表示没有错。
2. H 为奇并且 p_4 为奇，表示发生了 1 位可纠正错误。（如果仅 1 位出错，p_4 应当为奇。）
3. H 为偶并且 p_4 为奇，表示仅 p_4 位出错，而不是该字的其余部分，因此取反 p_4 位即可。
4. H 为奇并且 p_4 为偶，表示发生了 2 位错。（当出现 2 位错时，p_4 为偶。）

纠 1 位错、检 2 位错（SEC / DED）技术在当今服务器的内存中被广泛应用。方便的是，8 字节的数据块做 SEC/DED 刚好需要一个字节的额外开销，这就是为什么许多 DIMM 是 72 位宽。

详细阐述 为了计算 SEC 需要多少位，设 p 表示校验位的位数，d 表示数据位的位数，那么整个字为 $p+d$ 位。如果采用 p 位纠错位指示错误（字长为 $p+d$ 位），再加上没有错误的情况，那么需要：

$$2^p \geqslant p+d+1 \text{ 位，因此 } p \geqslant \log(p+d+1)$$

例如，对于 8 位的数据意味着 $d=8$ 并且 $2^p \geqslant p+8+1$，所以 $p=4$。类似地，16 位数据的 $p=5$，32 位时 p=6，64 位时 p=7，依此类推。

详细阐述 在大型系统中，出现多位错的概率和整个内存芯片出错的概率变得显著。IBM 引入叫作 chipkill 的技术来解决这个问题，许多大型系统也使用这种技术。（Intel 称他们所用的为 SDDC。）与用于磁盘的 RAID 方法类似（见 5.11 节），chipkill 将数据和校验码分散开，因此当某一内存芯片全部出错时，可以通过其他内存芯片对丢失的内容进行重建。假设有 10000 个处理器组成的集群，其中每个处理器有 4GiB 内存，IBM 计算了在 3 年运行中以下不可恢复的内存错误出现的比率：

- 仅采用奇偶校验：约有 90 000 次不可恢复（或不可检测）的错误，即每 17 分钟出现一次。
- 仅采用 SEC / DED：约有 3500 次不可恢复（或不可检测）的错误，即每 7.5 小时出现一次。
- 采用 chipkill：有 6 次不可恢复（或不可检测）的错误，即每两个月出现一次。

因此，chipkill 是仓储级计算机需要采用的技术。

详细阐述 虽然存储器系统出现 1 位错或 2 位错的情况比较典型，但网络系统中可能会出现突发性错误。一种解决方案称为循环冗余校验。对于一个 k 位的数据块，发送端产生一个 $n-k$ 位的帧校验序列。这样最终发送出 n 位序列，并且该序列构成的数字可以被某个数整除。接收端那个数去除接收到的帧，如果余数为 0，则认为没有错误；如果余数不为 0，则接收端拒绝该消息，并要求发射端再次发送。从第 3 章不难猜测到，使用移位寄存器可以很容易地计算某些二进制数的除法，这使得即使在硬件资源很珍贵的时代，CRC 码也被广泛使用。更进一步，里德－所罗门（Reed-Solomon）码使用伽罗瓦（Galois）域来纠正多位传输错误，数据被看作多项式的系数，校验码看作多项式的值。里德－所罗门的计算复杂度远高于二进制除法的复杂度！

5.6　虚拟机

虚拟机（virtual machine）最早出现于20世纪60年代中期，多年来一直是大型机的重要组成部分。尽管在20世纪80和90年代，虚拟机大多被单用户个人计算机所忽视。但由于以下几个因素，最近虚拟机又重新受到关注：

- 在现代计算机系统中，隔离和安全的重要性日益增加。
- 标准操作系统在安全性和可靠性方面存在缺陷。
- 许多不相关用户共享一台计算机，特别是云计算。
- 几十年来，处理器速度大幅增加，这使得虚拟机的开销变得可以接受。

最广泛的虚拟机的定义包括所有基本的仿真方法，这些方法提供标准软件接口，例如Java VM。本节介绍虚拟机如何在二进制指令系统体系结构（ISA）的层次上，提供一个完整的系统级环境。虽然有些虚拟机在本地硬件上运行不同的ISA，但我们假设它们都能与硬件匹配。这些虚拟机被称为（操作）系统虚拟机。例如IBM VM / 370、VirtualBox、VMware ESX Server和Xen。

系统虚拟机让用户觉得自己拥有包括操作系统副本在内的整台计算机。一台运行多个虚拟机的计算机可以支持多个不同的操作系统。在传统平台上，单个操作系统拥有所有硬件资源，但对于虚拟机，多个操作系统共享硬件资源。

支持虚拟机的软件被称为虚拟机监视器（VMM）或管理程序，VMM是虚拟机技术的核心。底层硬件平台被称为主机（host），它的资源被客户端（guest）虚拟机共享。VMM决定如何将虚拟资源映射到物理资源：物理资源可能是分时共享、划分甚至是软件模拟的。VMM比传统操作系统小得多，一个VMM的隔离区可能只有10 000行代码。

尽管我们所感兴趣的在于虚拟机能够提供保护功能，但虚拟机还有两个具有商业价值的优点：

1. 管理软件。虚拟机提供一个可以运行整个软件堆的抽象，甚至包括像DOS这样的旧操作系统。虚拟机的典型部署可能是：一些虚拟机运行旧的操作系统，多数虚拟机运行当前稳定的操作系统版本，少数虚拟机测试下一个操作系统版本。

2. 管理硬件。使用多台服务器的一个目的是使每个应用程序运行在一台独立的、有着与之兼容的操作系统的计算机上，因为这种分离可以提高可靠性。虚拟机使这些分离的软件堆独立运行，但共享硬件，从而合并了服务器的数量。另一个例子是，一些VMM支持将正在运行的VM移植到另一台计算机上，以平衡负载或在硬件发生故障时实施迁移。

硬件/软件接口　亚马逊网络服务（AWS）在其云计算平台中使用虚拟机提供EC2，有以下五个原因：

1. 多个用户共享相同的服务器时，AWS可保护用户免受彼此的影响。

2. 它简化了仓储级计算机上的软件分布。用户安装一个虚拟机镜像，并配置合适的软件，AWS为用户分配所需的虚拟机镜像。

3. 在用户完成工作时，用户（和AWS）可以可靠地“杀死”虚拟机以控制资源的使用。

4. 虚拟机隐藏了运行用户软件的硬件的特性，这意味着AWS可以继续使用旧服务器同时引入新的更高效的服务器。用户希望获得的性能与“EC2计算单元”匹配，AWS将其定义为“与1.0-1.2 GHz 2007 AMD Opteron或2007 Intel Xeon处理器相等的CPU能力”。根据摩尔定律，新的服务器显然能比旧的服务器提供更多的EC2计算单元，但只要经济实惠，

AWS 就可以继续出租旧服务器。

5.VMM 可以控制虚拟机使用处理器、网络和磁盘的比例，这使 AWS 可以在相同底层服务器上提供不同类型的服务，这些服务对应不同的价格。例如，2012 年 AWS 提供了从每小时 0.08 美元的小型标准服务到每小时 3.10 美元的高 I/O 特大型服务等 14 种服务。

一般来说，处理器虚拟化的开销取决于工作负载。用户级处理器绑定的程序没有虚拟化开销，因为操作系统很少被调用，所以一切都以本地速度运行。I/O 密集型工作负载通常也是操作系统密集型的，因为要执行很多系统调用和特权指令，从而导致很高的虚拟化开销。另一方面，如果 I/O 密集型工作负载也是受限于 I/O 的，那么处理器虚拟化的开销可以被完全隐藏，因为处理器通常处于空闲状态中等待 I/O。

开销取决于 VMM 要模拟的指令数量以及模拟每条指令需要多少时间。因此，当客户端虚拟机与主机运行相同的 ISA 时，正如我们假设的那样，体系结构和 VMM 的目标是尽可能直接在本地硬件上运行所有的指令。

5.6.1 虚拟机监视器的必备条件

虚拟机监视器需要做什么？它为客户端软件提供软件接口，隔离每个客户端的状态，并且必须保护自己免受客户端软件（包括客户端操作系统）的侵害。定性的需求是：

- 除了与性能相关的行为或由于多个 VM 共享所导致的固定资源的限制外，客户端软件应该像在本地硬件上一样在虚拟机上运行。
- 客户端软件不能直接改变实际系统资源的分配。

为了“虚拟化”处理器，VMM 必须控制几乎所有事情：特权态的访问、I/O、例外和中断——即使客户端虚拟机和当前运行的操作系统只是临时使用它们。

例如，在发生定时器中断时，VMM 将挂起当前正在运行的客户端虚拟机、保存状态、处理中断，确定下一个要运行的客户端虚拟机，并读取其状态。通过 VMM 为依赖于定时器中断的客户端虚拟机提供虚拟定时器并模拟定时器中断。

为方便管理，VMM 必须拥有比客户虚拟机更高的权限，其中用户虚拟机通常在用户模式运行，这也确保了任何特权指令的执行都将由 VMM 处理。支持 VMM 的基本系统要求是：

- 至少有两种处理器模式：系统模式和用户模式。
- 特权指令集合只能在系统模式下使用，如果在用户模式下执行将会导致内陷；所有系统资源只能通过这些指令控制。

5.6.2 指令系统体系结构（缺乏）对虚拟机的支持

如果在设计 ISA 时考虑到虚拟机，减少必须由 VMM 执行的指令数量并提高其仿真速度就相对容易。允许虚拟机直接在硬件上执行的体系结构被冠以可虚拟化的名称，IBM 370 和 RISC-V 体系结构就是如此。

由于虚拟机仅在最近才被考虑应用于 PC 和服务器应用程序，因此大多数指令系统在创建时没有考虑虚拟化。这些指令系统包括 x86 和大多数 RISC 架构（包括 ARMv7 和 MIPS）。

VMM 必须确保客户系统仅与虚拟资源交互，因此传统的客户操作系统在 VMM 顶层运行用户模式程序。如果客户操作系统试图通过特权指令访问或修改与硬件资源相关的信息（例如，读或写允许中断的状态位），那么它会陷入 VMM。然后，VMM 可以对相应实际资源

进行适当的更改。

因此，如果任何指令试图在用户模式下执行读或写这种敏感信息而产生自陷，VMM会拦截它并像客户操作系统所期望的那样，提供敏感信息的虚拟版本。

如果不提供上述支持，必须采取其他措施。VMM要采取特殊的预防措施来定位所有有问题的指令，并确保它们能被客户操作系统正确执行，这在增加VMM的复杂性同时也降低了VM的运行性能。

5.6.3　保护和指令系统体系结构

保护需要依靠体系结构和操作系统的共同努力，但是随着虚拟存储的广泛使用，体系结构设计者不得不修改现有指令系统体系结构中一些不方便的细节。

例如，x86指令POPF执行从存储器中的堆栈顶部加载数据至标志寄存器。其中一个标志是中断使能（IE）标志。如果在用户模式下执行POPF指令，不会发生内陷，而是会改变除IE以外的所有标志位。如果在系统模式下，这条指令确实会改变IE位。但是有一个问题，运行在虚拟机用户模式下的客户操作系统希望看到IE位的改变。

历史上，IBM大型机硬件和VMM采取以下三个步骤来提高虚拟机性能：

1. 降低处理器虚拟化的开销。
2. 降低由虚拟化引起的中断开销。
3. 将中断交给相应的VM而不调用VMM，从而降低中断开销。

在2006年，AMD和Intel提出新的计划尽力满足第一点，即降低处理器虚拟化的开销。体系结构和VMM需要经过多少代的改进才能完全满足上面三点？21世纪的虚拟机需要经过多长时间才能像20世纪70年代的IBM大型机VMM一样有效？这些都是令人感兴趣的研究。

详细阐述　在用户模式下运行时，RISC-V可以使所有特权指令内陷，因此它支持传统虚拟化，其中客户操作系统以用户模式运行，VMM以管理模式运行。

5.7　虚拟存储

在前面的章节中，我们知道了cache如何对程序中最近访问的代码和数据提供快速访问。同样，主存可以为通常由磁盘实现的辅助存储充当“cache”。这种技术被称为**虚拟存储**。从历史上看，提出虚拟存储的主要动机有两个：允许在多个程序之间高效安全地共享内存，例如云计算的多个虚拟机所需的内存，以及消除小而受限的主存容量对程序设计造成的影响。50年后，第一条变成主要设计动机。

现在的系统是这样的：对程序员来说，组合的存储结构看起来像单层的存储，其中必要的转换是自动发生的。
Kilburn等，One-Level Storage System，1962

当然，为了允许多个虚拟机共享内存，必须保护虚拟机免受其他虚拟机影响，确保程序只读写分配给它的那部分主存。主存只需存储众多虚拟机中活跃的部分，就像cache只存放一个程序的活跃部分一样。因此，局部性原理支持虚拟存储和cache，虚拟存储允许我们高效地共享处理器以及主存。

虚拟存储：一种将主存看作辅助存储的cache技术。

在编译虚拟机时，无法知道哪些虚拟机将与其他虚拟机共享存储。事实上，共享存储的

虚拟机在运行时会动态变化。由于这种动态的交互作用，我们希望将每个程序都编译到它自己的地址空间中——只有这个程序能访问的一系列存储位置。虚拟存储实现了将程序地址空间转换为**物理地址**。这种地址转换处理加强了各个程序地址空间之间的**保护**。

物理地址：主存的地址。

保护：一组保护机制，确保共享处理器、内存、I/O设备的多个进程之间没有故意地、无意地读写其他进程，这些保护机制可以将操作系统和用户的进程隔离开来。

虚拟内存的第二个动机是允许单用户程序使用超出内存容量的内存。以前，如果一个程序对于存储器来说太大，程序员应该调整它。程序员将程序划分成很多段，并将这些段标记为互斥的。这些程序段在执行时由用户程序控制载入或换出，程序员确保程序在任何时候都不会访问未载入的程序段，并且载入的程序段不会超过内存的总容量。传统的程序段被组织为模块，每个模块都包含代码和数据。不同模块之间的过程调用将导致一个模块覆盖掉另一个模块。

缺页失效：被访问的页不在主存中的事件。

可以想象，这种责任对程序员来说是很大的负担。虚拟存储的发明就是为了将程序员从这些困境中解脱出来，它自动管理由主存（有时称为物理内存，以区分虚拟存储）和辅助存储所代表的两级存储层次结构。

虚拟地址：虚拟空间的地址，当访问内存时需要通过地址映射转换为物理地址。

虽然虚拟存储和 cache 的工作原理相同，但不同的历史根源导致它们使用的术语不同。虚拟存储块被称为页，虚拟存储失效被称为缺页失效。在虚拟存储中，处理器产生一个**虚拟地址**，该地址通过软硬件转换为一个物理地址，物理地址可访问主存。图 5-25 显示了分页的虚拟存储被映射到主存中。这个过程被称为地址映射或**地址转换**。如今，由虚拟存储控制的两级存储层次结构通常是个人移动设备中的 DRAM 和闪存，在服务器中是 DRAM 和磁盘（见 5.2 节）。如果回到我们的图书馆类比，可以将虚拟地址视为书名，将物理地址视为图书馆中该书的位置，它可能是图书馆的索书号。

地址转换：也称为地址映射。访问内存时将虚拟地址映射为物理地址的过程。

虚拟内存还通过重定位简化了执行时程序的载入。在用地址访问存储之前，重定位将程序使用的虚拟地址映射到不同的物理地址。重定位允许将程序载入主存中的任何位置。此外，现今所有的虚拟存储系统都将程序重定位成一组固定大小的块（页），从而不需要寻找连续内存块来放置程序；相反，操作系统只需要在主存中找到足够数量的页。

在虚拟存储中，地址被划分为虚拟页号和页内偏移。图 5-26 显示了虚拟页号到物理页号的转换。虽然 RISC-V 有 64 位地址，但其中的高 16 位未使用，因此要映射的地址有 48 位。假定物理内存容量为 1TiB 或 2^{40} 字节，则需要 40 位地址。物理页号构成物理地址的高位部分。页内偏移不变，它构成物理地址的低位部分。页内偏移的位数决定页的大小。虚拟地址可寻址的页数与物理地址可寻址的页数可以不同。拥有比物理页更多数量的虚拟页是一个没有容量限制的虚拟存储的基础。

缺页失效的高成本是许多设计选择虚拟存储系统的原因。磁盘的缺页失效处理需要数百万个时钟周期。（图 5-5 显示主存延迟比磁盘快大约 10 万倍。）这种巨大的失效代价（主要由获得标准页大小的第一个字所花费的时间来确定）导致了设计虚拟存储系统时的几个关键决策：

- 页应该足够大以分摊长访问时间。目前典型的页大小从 4KiB 到 64KiB 不等。支持 32KiB 和 64KiB 页的新型桌面和服务器正在研发，但是新的嵌入式系统正朝另一个方向发展，页大小为 1KiB。

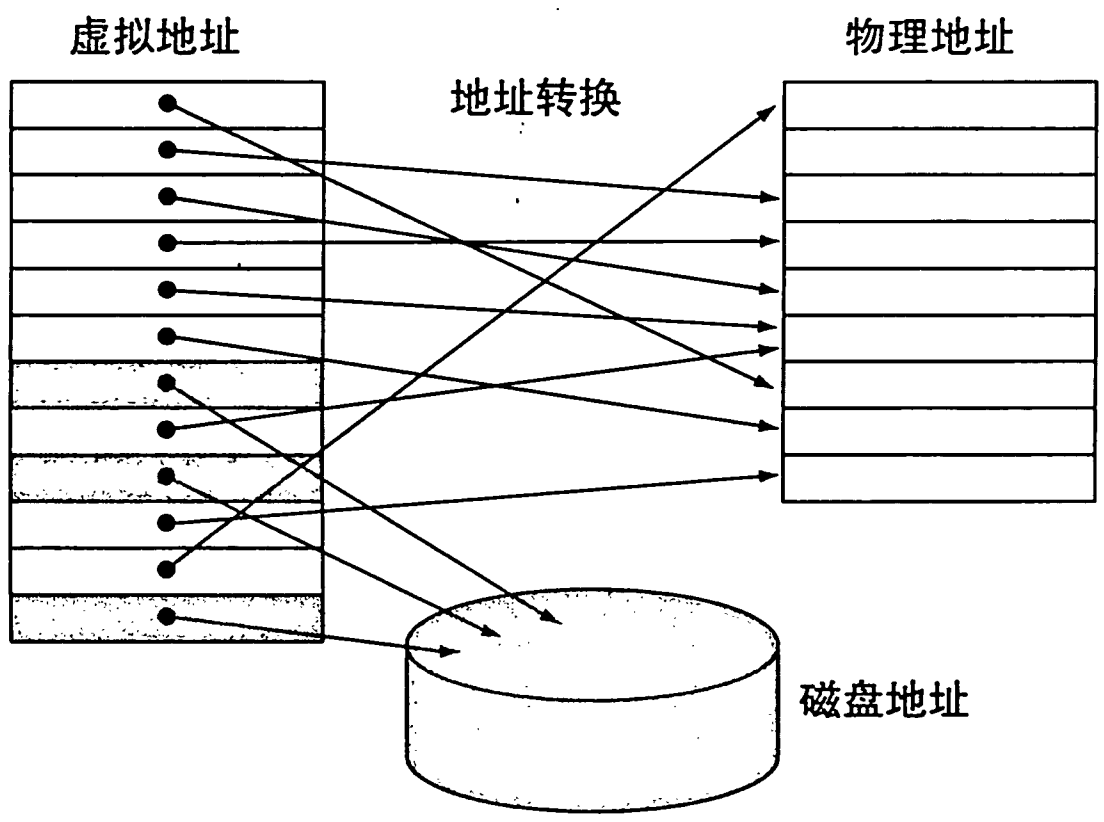

图 5-25 在虚拟内存中，内存块（称为页）从一组地址（称为虚拟地址）映射到另一组（称为物理地址）。处理器产生虚拟地址，而使用物理地址访问内存。虚拟存储和物理存储都被划分为页，因此一个虚拟页被映射到一个物理页。当然，虚拟页也可能不在主存中，因此不能映射到物理地址；在这种情况下，页被存在磁盘上。物理页也可以被两个指向相同物理地址的虚拟地址共享，用于使两个不同的程序共享数据或代码

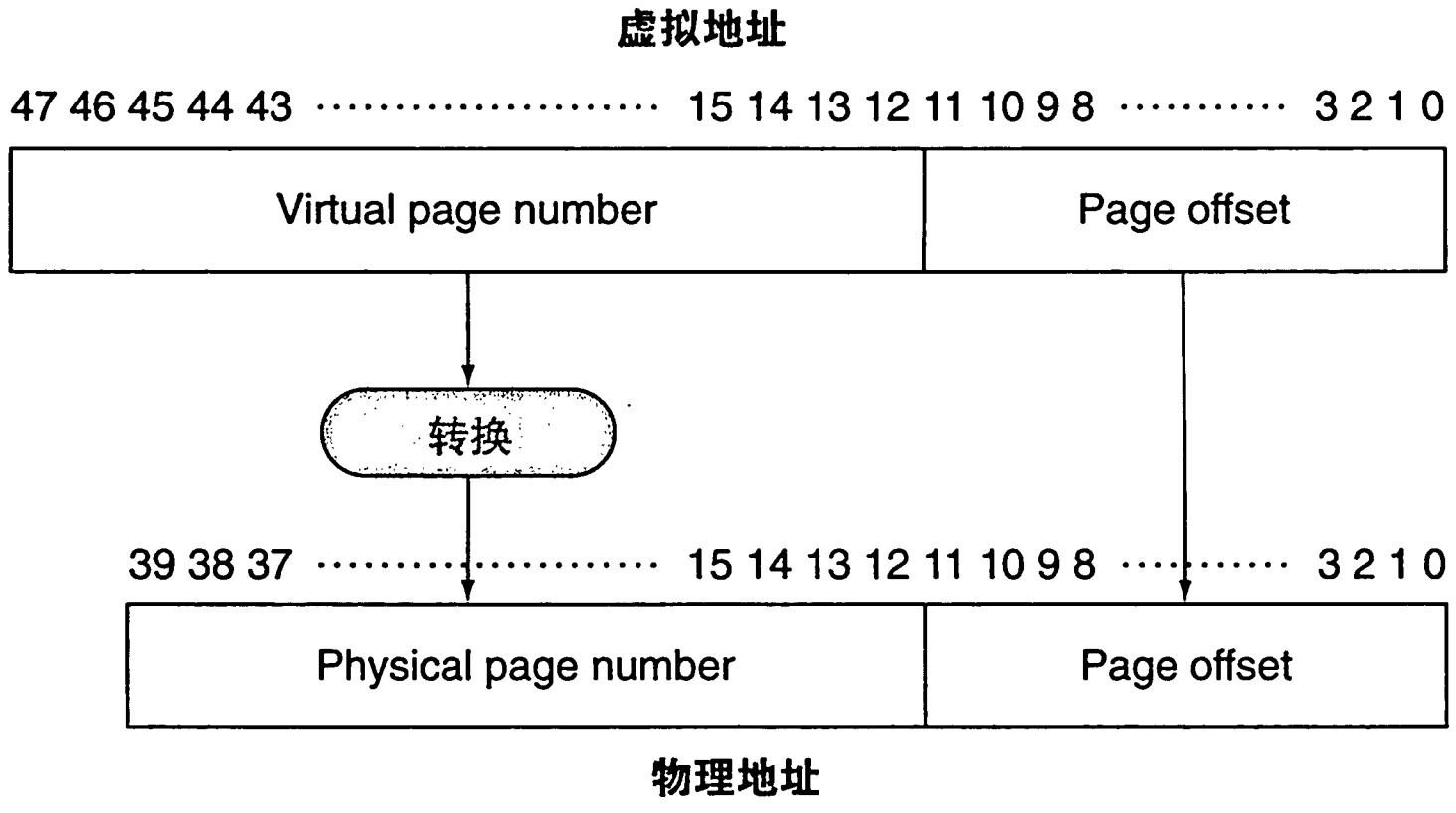

图 5-26 虚拟地址到物理地址的映射。页大小为 2^{12}=4KiB。由于物理页号有 28 位，内存中物理页数为 2^{28}。因此，主存最大容量为 1TiB，而虚拟地址空间为 256TiB。RISC-V 允许物理内存容量达到 1PiB，我们选择 1TiB 是因为这对于 2016 年的很多计算机而言已经足够

- 能降低缺页失效率的组织结构很有吸引力。这里使用的主要技术是允许存储中的页以全相联方式放置。
- 缺页失效可以由软件处理，因为与磁盘访问时间相比，这样的开销将很小。此外，软件可以用巧妙的算法来选择如何放置页面，只要失效率减少很小一部分就足以弥补算法的开销。
- 写穿透策略对于虚拟存储不合适，因为写入时间过长。相反，虚拟存储系统采用写回策略。

下面几节将把这些因素融入虚拟存储的设计中去。

详细阐述 我们说明了虚拟存储的动机：为了允许许多虚拟机共享相同的存储。但设计虚拟存储最初的原因是为了在分时系统中，让许多程序可以共享一台计算机。由于当今很多

读者没有使用分时系统的经验，本节使用虚拟机作为引入虚拟存储的动机。

详细阐述 RISC-V 支持各种虚拟存储的配置。除了适用于 2016 年的大型服务器的 48 位虚拟地址方案之外，该架构还可以支持 39 位和 57 位的虚拟地址。所有这些配置都使用 4 KiB 大小的页面。

详细阐述 对于服务器甚至个人电脑，32 位地址的处理器已经很有问题了。通常我们认为虚拟地址比物理地址大得多，但当处理器地址宽度相对于存储器技术而言较小时，可能会出现相反的情况。单个程序或虚拟机不会受益，但同时运行的一组程序或虚拟机可能由于无须从主存中换出，或可以在并行处理器上运行而受益。

详细阐述 本书中有关虚拟存储的讨论集中于页式存储，即使用固定大小的块。还有一种可变长度块的机制称为段式存储。在段式存储中，地址由两部分组成：段号和段内偏移。段号被映射到物理地址，与段内偏移相加得到实际的物理地址。由于段的大小可变，因此还需要进行边界检查以确定偏移量是否在段内。分段的主要应用是支持更强大的保护和地址空间的共享。大多数操作系统教科书更多地讨论分段，以及如何利用分段来从逻辑上共享地址空间。分段的主要缺点是将地址空间分割成逻辑上独立的部分，因此这些块就由两部分地址控制——段号和段内偏移。相比而言，分页使得页号和页内偏移的界限对于程序员和编译器都不可见。

段式存储：一种可变长度的地址映射策略，其中每个地址由两部分组成：映射到物理地址的段号和段内偏移。

分段也曾被用作不改变计算机字长而扩展地址空间的方法。然而这些尝试并未成功，这是由于程序员和编译器必须意识到使用两部分地址所带来的不便和性能损失。

许多体系结构将地址空间划分为固定大小的大块以简化操作系统和用户程序之间的保护，并提高分页实现的效率。虽然这些划分通常被称为“段”，但这种机制比可变大小的分段简单得多，并且对用户程序不可见。稍后将详细讨论。

5.7.1 页的存放和查找

由于缺页失效的代价非常高，设计人员通过优化页的放置来降低缺页失效频率。如果允许一个虚拟页映射到任何一个物理页，那么在发生缺页失效时，操作系统可以选择任意一个页进行替换。例如，操作系统可以使用复杂的算法和复杂的数据结构来跟踪页面使用情况，来选择在较长一段时间内不会被用到的页。使用先进而灵活的替换策略降低了缺页失效率，并简化了全相联方式下页的放置。

正如 5.4 节所述，全相联映射的困难在于项的定位，因为它可以在较高存储层次结构中的任何位置。全部进行检索是不切实际的。在虚拟存储系统中，我们使用一个索引主存的表来定位页；这个结构称为**页表**，它被存在主存中。页表使用虚拟地址中的页号作为索引，找到相应的物理页号。每个程序都有自己的页表，它将程序的虚拟地址空间映射到主存。在图书馆类比中，页表对应于书名和图书馆位置之间的映射。就像卡片目录可能包含学校中另一个图书馆中书的表项，而不仅仅是本地的分馆，我们将看到该页表也可能包含不在内存中的页的表项。为了表明页表在内存中的位置，硬件包含一个指向页表首地址的寄存器，我们称之为页表寄存器。现在假定页表存在存储器中一个固定的连续区域中。

页表：在虚拟存储系统中，保存着虚拟地址和物理地址之间转换关系的表。页表保存在内存中，通常使用虚拟页号来索引，如果这个页在内存中，页表中的对应项包含该页对应的物理页号。

硬件/软件接口 页表与程序计数器和寄存器一起确定了一个虚拟机的状态。如果我们

想让另一个虚拟机使用处理器，必须保存这个状态。在恢复此状态后，虚拟机可以继续执行。我们通常称这个状态为一个进程。如果一个进程占用了处理器，那么这个进程是活跃的；否则是非活跃的。操作系统可以通过载入进程的状态来激活进程，包括程序计数器，进程将从程序计数器中保存的值处开始执行。

进程的地址空间，以及它在内存中可以访问的所有数据，都由其存储在内存中的页表定义。操作系统并不保存整个页表，而是简单地载入页表寄存器来指向它想要激活的进程的页表。由于不同的进程使用相同的虚拟地址，因此每个进程都有各自的页表。操作系统负责分配物理内存并更新页表，这样不同进程的虚拟地址空间不会发生冲突。我们很快会看到，使用分离的页表也可以分别保护进程。

图 5-27 使用页表寄存器、虚拟地址和被指向的页表来说明硬件如何形成物理地址。正如在 cache 中所做的那样，每个页表项中使用 1 位有效位。如果该位为无效，则该页就不在主存中，发生一次缺页失效。如果该位为有效，则该页在内存中，并且该项包含物理页号。

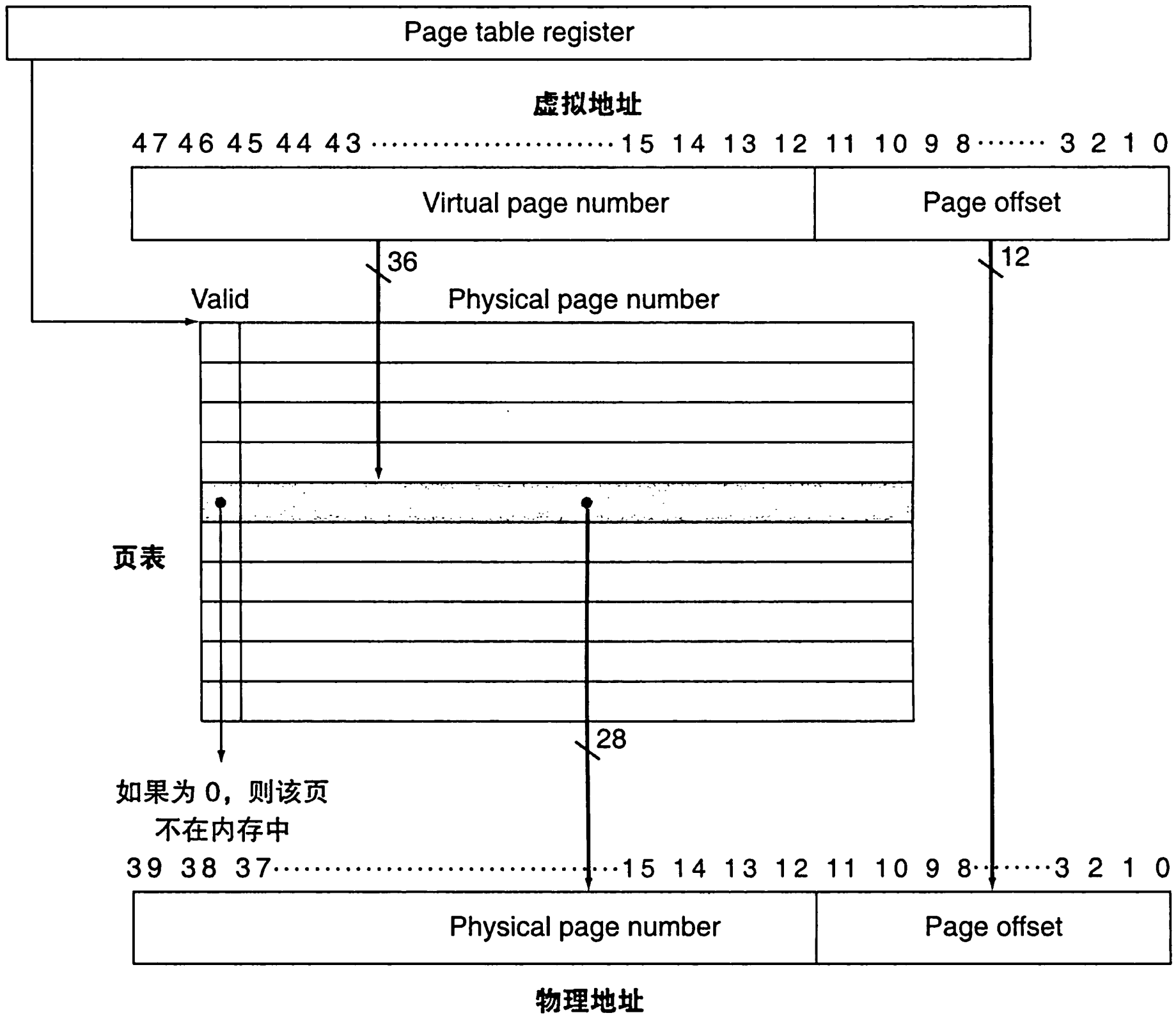

图 5-27　使用虚拟页号索引页表以获得对应的物理地址部分。假设地址为 48 位。页表指针给出了页表的起始地址。在该图中，页大小为 2^{12} 个字节，即 4KiB。虚拟地址空间为 2^{48} 字节或 256TiB，物理地址空间为 2^{40} 字节，主存最大容量为 1TiB。如果 RISC-V 使用如本图所示的一张页表，则页表中的表项数将为 2^{36}，即大约 640 亿个表项。（我们将看到 RISC-V 会尽快减少表项数。）每个表项的有效位指示映射是否合法。如果它为 0，则该页不在内存中。尽管图中显示的页表表项只需 29 位宽，但为了寻址方便，通常让它舍入到 2 的幂次位。RISC-V 中的页表项是 64 位宽。其他位则用来存放每页都要保留的基本的附加信息，如保护信息

由于页表包含了每个可能的虚拟页的映射，因此不需要标签。在 cache 术语中，索引是用来访问页表的，这里由整个块地址即虚拟页号组成。

5.7.2　缺页失效

如果虚拟页的有效位为无效，则会发生缺页失效。操作系统获得控制。这种控制的转移通过例外机制完成，在第 4 章中我们已经了解了例外机制，本节稍后将再次讨论。一旦操作系统得到控制，它必须在存储层次结构的下一级（通常是闪存或磁盘）中找到该页，并确定将请求的页放在主存中的什么位置。

虚拟地址本身并不会立即告诉我们该页在辅助存储中的位置。回到图书馆的类比，我们无法仅依靠书名就找到图书的具体位置。而是按目录查找，获得书在书架上的位置信息，比如说图书馆的索引书号。同样，在虚拟存储系统中，我们必须跟踪记录虚拟地址空间的每一页在辅助存储中的位置。

交换区：为进程的全部虚拟地址空间所预留的磁盘空间。

由于我们无法提前获知存储器中的某一页什么时候将被替换出去，因此操作系统通常会在创建进程时为所有页面在闪存或磁盘上创建空间。这个空间被称为**交换区**。那时，它也会创建一个数据结构来记录每个虚拟页在磁盘上的存储位置。该数据结构可以是页表的一部分，或者可以是具有与页表相同索引方式的辅助数据结构。图 5-28 显示了一个保存物理页号或辅助存储器地址的单个表的结构。

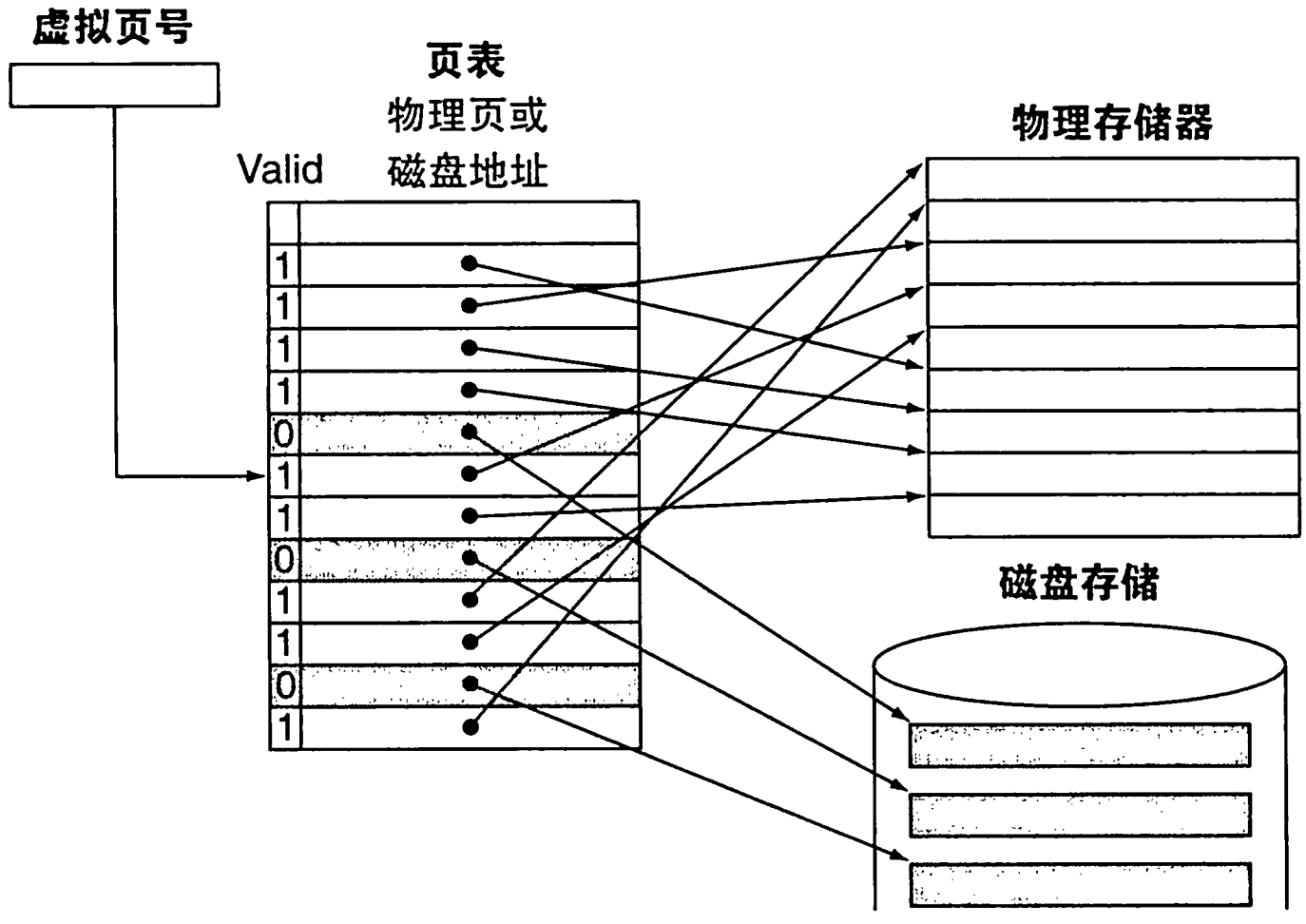

图 5-28　页表将虚拟存储器中的每一页映射到内存中的一页或者存储层次结构下一层的一页（磁盘上的一页）。虚拟页号用来检索页表。如果有效位为有效，页表提供虚拟页对应的物理页号（如内存中该页的首地址）。如果有效位为无效，那么该页就只存在磁盘上的某个指定的磁盘地址。在许多系统中，物理页地址和磁盘页地址的表在逻辑上是一个表，但是保存在两个独立的数据结构中。因为即使有些页当前不在内存中，也必须保存所有页的磁盘地址，所以使用双表在某种程度上是合理的。请记住内存中的页和磁盘上的页大小相等

操作系统还会创建一个数据结构用于跟踪记录使用每个物理地址的是哪些进程和哪些虚拟地址。发生缺页失效时，如果内存中的所有页都正在使用，则操作系统必须选择一页进行替换。因为我们希望尽量减少缺页失效次数，所以大多数操作系统选择它们认为近期内不会

使用的页进行替换。使用过去的信息预测未来，操作系统遵循在5.4节中提到的最近最少使用（LRU）替换策略。操作系统查找最近最少使用的页，假定某一页在很长一段时间都没有被访问，那么该页再被访问的可能性比最近经常访问的页的可能性要小。被替换的页被写到辅助存储器中的交换区。如果还不是很明白，可以把操作系统看成另一个进程，而那些控制内存的表也在内存中；这看起来似乎有些矛盾，稍后将具体解释。

引用位：也称为使用位。每当访问一个页面时该位被置位，通常用来实现LRU或其他替换策略

硬件/软件接口 要完全准确地执行LRU算法的代价太高了，因为每次存储器访问时都需要更新数据结构。作为替代，大多数操作系统通过跟踪哪些页最近被使用，哪些页最近没有用到来近似地实现LRU算法。为了帮助操作系统估算最近最少使用的页，RISC-V计算机提供了一个引用位（reference bit）或者称为使用位（use bit），当一页被访问时该位被置位。操作系统定期将引用位清零，然后再重新记录，这样就可以判定在这段特定时间内哪些页被访问过。有了这些使用信息，操作系统就可以从那些最近最少访问的页中选择一页（通过检查其引用位是否关闭）。如果硬件没有提供这一位，操作系统就要通过其他的方法来估计哪些页被访问过。

5.7.3 支持大虚拟地址空间的虚拟存储

图5-27的标题中指出，对于4KiB大小的页，如果虚拟地址是48位，并采用单级页表，则页表有640亿（2^{36}）个表项。由于RISC-V每个页表项中为8个字节，因此将虚拟地址映射到物理地址就需要0.5TiB存储空间！而且，如果计算机中同时有成百的进程同时运行，每个进程都有自己的页表。即使对于最大的系统，这样大量的内存也无法承受。

一系列技术已经被用于减少页表所需的存储空间。以下五种技术从两个角度解决问题，一是减少存放页表所需的最大存储空间，二是减少用于存放页表的存储空间。

1. 最简单的技术是保留界限寄存器，限制给定进程的页表大小。如果虚拟页号大于界限寄存器的值，那么表项将被添加到页表中。这种技术允许页表随着进程消耗空间的增多而增长。因此，只有当进程使用了许多虚拟页时，页表才会变得很大。这种技术要求地址空间只向一个方向扩展。

2. 允许地址空间只朝一个方向增长并不够，因为大多数语言都需要两个大小可扩展的区域：一个区域容纳栈，另一个区域容纳堆。由于这种二元性，划分页表并使其可以从最高地址向下增长，也可以从最低地址向上增长就方便多了。这意味着将有两个单独的页表和两个单独的界限寄存器。两个页表的使用将地址空间分成两段。地址的高位决定为该地址使用哪个段以及哪个页表。由于高位地址位指定段，因此每个段的容量可以等于一半的地址空间。每个段的界限寄存器指定当前段的大小，并以页为单位增长。与5.7节第二个“详细阐述”中讨论的分段不同，这种形式的段对应用程序是不可见的，但是对操作系统可见。这种方案的主要缺点是：当地址空间以稀疏方式而不是连续方式被访问时，它不能很好地工作。

3. 另外一种减小页表容量的方法是对虚拟地址应用哈希函数，以使页表容量等于主存中物理页的数量。这种结构称为反向页表。当然，使用反向页表的查找过程稍微复杂一些，因为我们不能只通过索引来访问页表。

4. 为减少页表占用的实际主存，大多数现代系统还允许将页表再分页。虽然这听起来很复杂，但是它的工作原理与虚拟存储相同，即允许将页表保存在虚拟地址空间中。另外，还必须避免一些小但很关键的问题，比如无止境的缺页失效。如何克服这些问题需要描述得很

详细，并且这些问题一般对机器的依赖性很高。简而言之，为了解决这些问题，可以将所有页表放置在操作系统的地址空间中，并至少将操作系统的一些页表放置在物理地址空间中且总是存在于主存而不出现在二级存储中。

5. 多级页表也可以用来减少页表存储的总量，这是 RISC-V 为减少地址转换占用的内存所采用的解决方案。图 5-29 显示了从 48 位虚拟地址到 4 KiB 页的 40 位物理地址的四级地址转换。首先使用地址的最高几位查看 0 级页表中的地址，进行地址转换。如果页表中对应的表项有效，则使用下一组高位地址来索引第 0 级页表表项指示的页表，依此类推。因此，第 0 级页表将虚拟地址映射到 512GiB（2^{39} 字节）区域。第 1 级页表又将虚拟地址映射到 1GiB（2^{30}）区域。再下一级页表将其映射到 2MiB（2^{21}）区域。最后一级页表将虚拟地址映射到 4 KiB（2^{12}）大小的内存页。该方案允许以稀疏的方式访问地址空间（多个不连续的段可以同时处于活跃状态），而不必分配整个页表。这种方案对于非常大的地址空间和需要非连续地址分配的软件系统特别有用。多级页表的主要缺点是地址转换过程复杂。

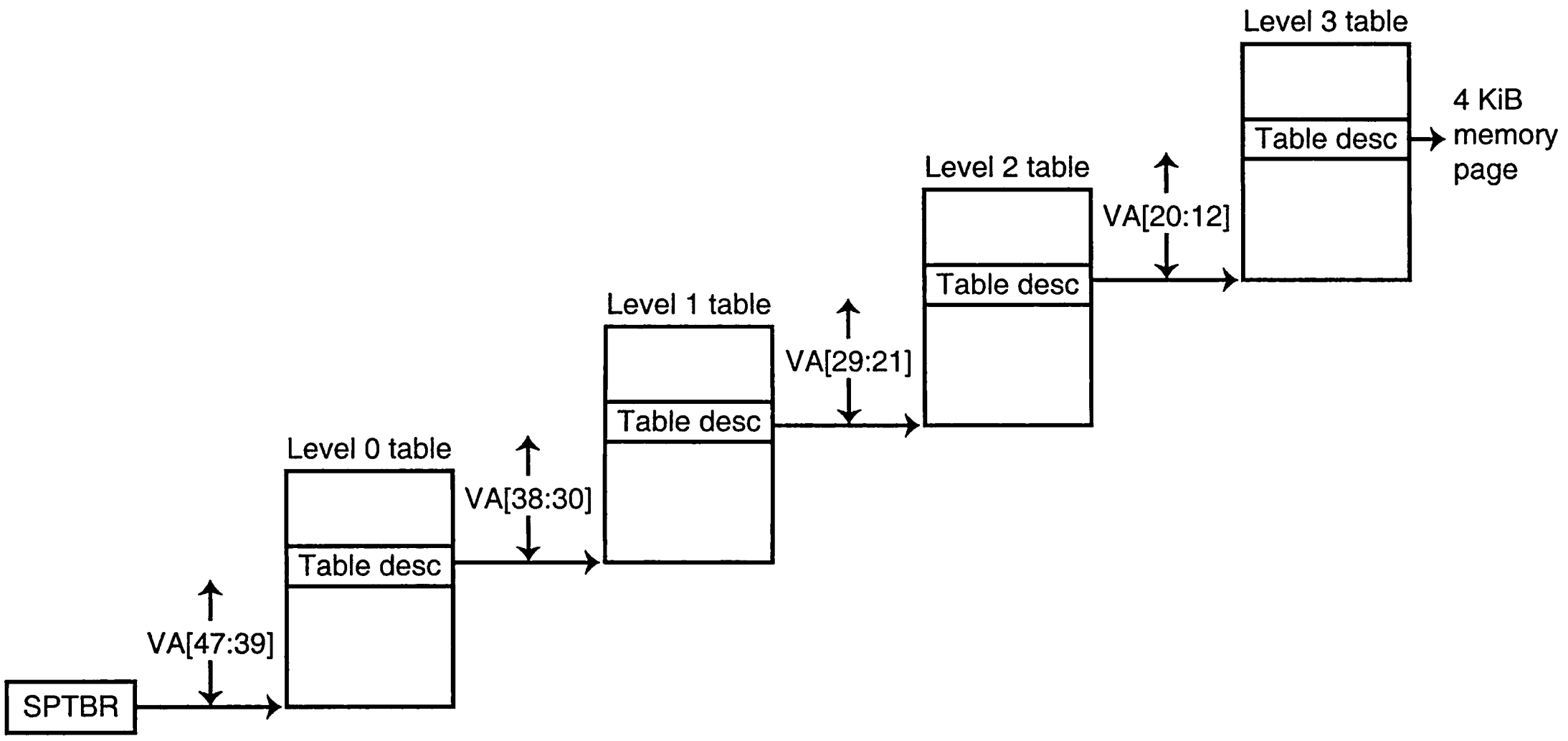

图 5-29 RISC-V 使用四级表将 48 位虚拟地址转换为 40 位物理地址。这种分层方法中页表很小，不像图 5-27 中的单个页面就需要 640 亿个页表项。转换的每一步都使用 9 位虚拟地址来查找下一级表，直到虚拟地址的高 36 位映射到对应 4KiB 页的物理地址。每个 RISC-V 页表项都是 8Byte 宽，所以表的 512 个表项可以填充一个 4KiB 的页。管理员页表基址寄存器（SPTIBR）给出了第一个页表的起始地址

5.7.4 关于写

访问 cache 和主存的时间相差几十到几百个时钟周期，如果采用写穿透策略，则需要一个写缓冲区向处理器隐藏写操作的延迟。在虚拟存储系统中，写入存储层次结构的下一级（磁盘）可能需要数百万个处理器时钟周期；因此，如果创建一个写缓冲区，允许系统采用写穿透的方式以对磁盘进行写的方法是完全不可行的。相反，虚拟存储系统必须使用写回策略，对内存中的页进行单独的写操作，并当页被从主存中替换出时，将其复制到辅助存储。

硬件/软件接口 写回策略在虚拟存储系统中还有一个重要优点。由于磁盘传输时间小于其访问时间，因此，将整个页复制回磁盘的效率高于将单个字写回。虽然写回操作比传输单独的字更快，但开销却很大。因此，在选择替换页时，我们希望知道是否需要复制该页。

为了追踪页面从被读入内存以后是否被写过，向页表中添加一个脏位。当一页中任何字被写入时，脏位被置位。如果操作系统选择要替换某一页，脏位指示在该页所占内存让给另一页之前，是否需要将该页写回磁盘。因此，修改后的页也通常称为脏页。

5.7.5　加快地址转换：TLB

由于页表存储在主存中，因此程序每个访存请求至少需要两次访存：第一次访存获得物理地址，第二次访存获得数据。提高访问性能的关键在于页表的访问局部性。当使用虚拟页号进行地址转换时，它可能很快再次被用到，因为对该页中字的引用同时具有时间局部性和空间局部性。

快表：用于记录最近使用地址的映射信息的cache，从而可以避免每次都要访问页表。

因此，现代处理器包含一个特殊的cache以追踪记录最近使用过的地址转换。这个特殊的地址转换cache通常被称为**快表**（TLB）（将其称为地址转换cache会更加准确）。TLB就相当于记录目录中的一些书的位置的小纸片：我们在纸片上记录一些书的位置，并且将小纸片当成图书馆索书号的cache，这样就不用一直在整个目录中搜索了。

图5-30显示TLB中的每个标签项保存虚拟页号的一部分，每个数据项保存一个物理页号。因为每次引用都访问TLB而不是页表，所以TLB需要包括其他状态位，例如脏位和引用位。虽然图5-30只显示了一张页表，但TLB也适用于多级页表。TLB从最后一级页表中载入物理地址和保护标签即可。

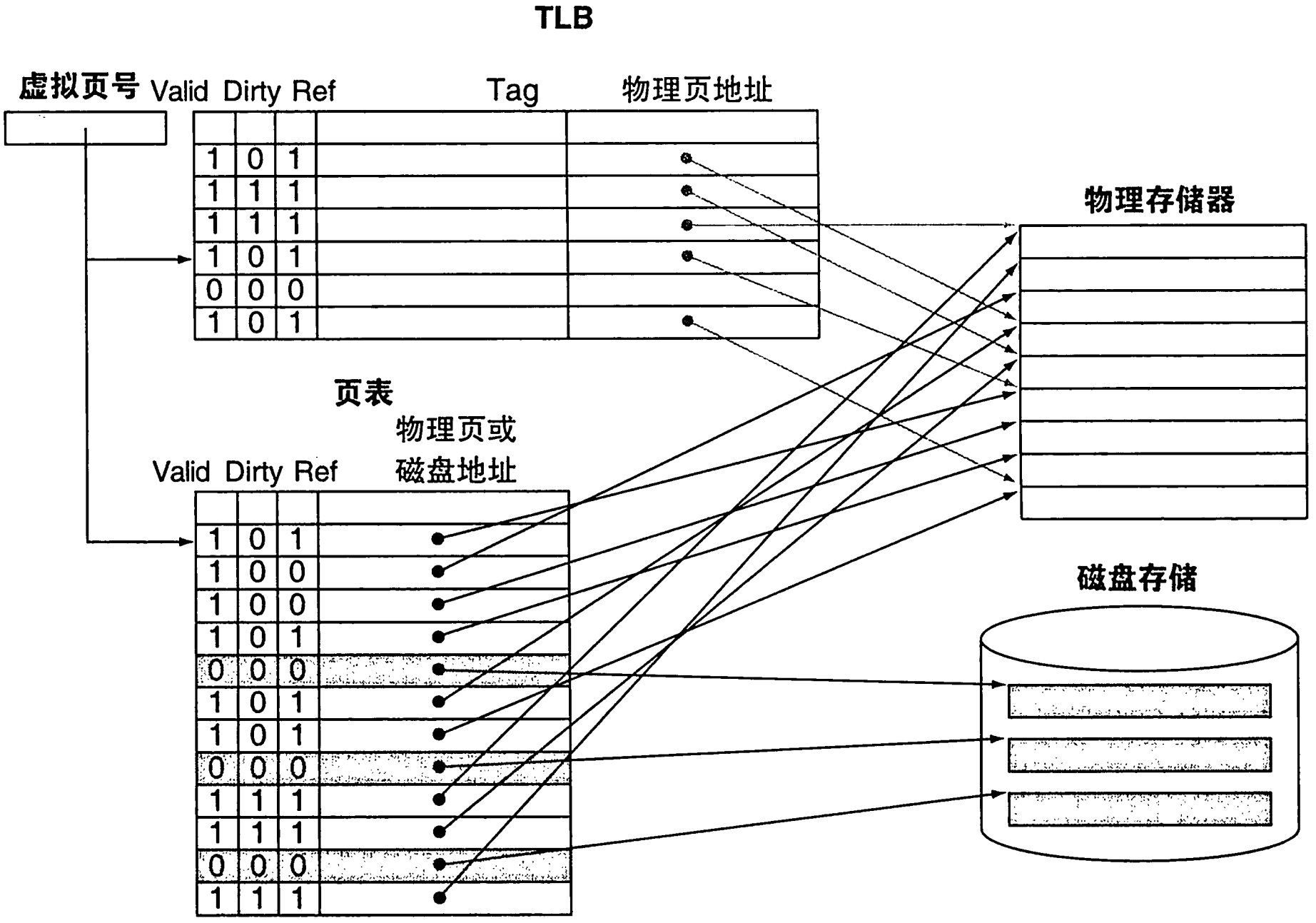

图5-30　TLB作为页表的cache，用于存放映射到物理页中的那些项。TLB包含页表中的虚拟页到物理页映射的一个子集。TLB映射以灰色显示。因为TLB是cache，它必须包含标签字段。如果一个页在TLB中没有匹配的项，则必须检查页表。页表提供页的物理页号（可用于创建TLB表项）或指示该页在磁盘上，这时就发生缺页失效。由于页表包含每个虚拟页的表项，因此不需要标签字段；换句话说，与TLB不同，页表不是cache

每次引用时，在 TLB 中查找虚拟页号。如果命中，则使用物理页号来形成地址，相应的引用位被置位。如果处理器要执行写操作，那么脏位也会被置位。如果 TLB 发生失效，我们必须确定是缺页失效或只是 TLB 失效。如果该页在内存中，TLB 失效表明缺少该地址转换。在这种情况下，处理器可以将（最后一级）页表中的地址转换加载到 TLB 中，并重新访问来处理失效。如果该页不在内存中，那么 TLB 失效意味着真正的缺页失效。在这种情况下，处理器调用操作系统的例外处理。由于 TLB 的项数比主存中的页数少得多，TLB 失效比缺页失效更频繁。

TLB 失效可以通过硬件或软件处理。实际上，两种方法之间几乎没有性能差异，因为它们的基本操作相同。

发生 TLB 失效并从页表中检索到失效的地址转换后，需要选择要替换的 TLB 表项。由于 TLB 表项包含引用位和脏位，所以当替换某一 TLB 表项时，需要将这些位复制回对应的页表项。这些位是 TLB 表项中唯一可修改的部分。使用写回策略——在失效时将这些表项写回而不是任何写操作都写回——是非常有效的，因为我们期望 TLB 失效率很低。一些系统使用其他技术来粗略估计引用位和脏位，这样在失效时无须写入 TLB，只需载入新的表项。

TLB 的一些典型值可能是

- TLB 大小：16 ～ 512 个表项
- 块大小：1 ～ 2 个页表项（通常每个为 4 ～ 8 字节）
- 命中时间：0.5 ～ 1 个时钟周期
- 失效代价：10 ～ 100 个时钟周期
- 失效率：0.01% ～ 1%

TLB 中相联度的设置非常多样。一些系统使用小的全相联 TLB，因为全相联映射的失效率较低；同时由于 TLB 很小，全相联映射的成本也不是太高。其他一些系统使用容量大的 TLB，通常其相联度较小。在全相联映射的方式下，由于硬件实现 LRU 方案成本太高，替换表项的选择就很复杂。此外，由于 TLB 失效比缺页失效更频繁，因此需要用较低的代价来处理，而不能像缺页失效那样选择一个开销大的软件算法。所以很多系统都支持随机选择替换表项。在 5.8 节将详细地介绍替换策略。

5.7.6 Intrinsity FastMATH TLB

为了在真实处理器中弄清楚这些想法，我们来仔细研究 Intrinsity FastMATH TLB。存储系统采用 4KiB 页面和 32 位地址空间，因此，虚拟页号长度为 20 位。物理地址与虚拟地址的宽度相同。TLB 包含 16 个表项，它是全相联的并且由指令和数据共享。每个表项为 64 位宽，包含一个 20 位的标签（该 TLB 表项的虚拟页号）、对应的物理页号（也是 20 位）、一个有效位、一个脏位以及一些其他管理操作位。像大多数 MIPS 系统一样，它用软件来处理 TLB 失效。

图 5-31 显示了 TLB 和一个 cache，图 5-32 显示了处理一次读或写请求的步骤。当发生 TLB 失效时，硬件将引用的页号保存在特殊寄存器中并产生例外。该例外调用操作系统，由软件处理失效。为了找到失效页的物理地址，TLB 失效程序将使用虚拟页号和页表寄存器来索引页表（页表寄存器指示活跃进程页表的起始地址）。通过执行更新 TLB 的一组特殊系统指令，操作系统将页表中的物理地址放入 TLB。假设代码和页表项分别位于指令 cache 和数据 cache 中，处理一次 TLB 失效大约需要 13 个时钟周期。如果页表项中的物理地址无效，则会发生真正的缺页失效。硬件保存着被建议替换项的索引，而这一项是随机选取的。