strikefreedom.top

虚拟内存精粹 - Strike Freedom

潘少潘少 Live fast. Die young. Be wild. Have fun.

35-44 minutes

文章内容上次编辑时间于 **2 月前**。文章距上次编辑时间较远，部分内容可能已经过时！

文章共 **8,127** 字，阅读完预计需要 **13 分钟 33 秒**。文章内容较长，请提前准备好咖啡!

导言

虚拟内存是当今计算机系统中最重要的抽象概念之一，它的提出是为了更加有效地管理内存并且降低内存出错的概率。虚拟内存影响着计算机的方方面面，包括硬件设计、文件系统、共享对象和进程/线程调度等等，每一个致力于编写高效且出错概率低的程序的程序员都应该深入学习虚拟内存。

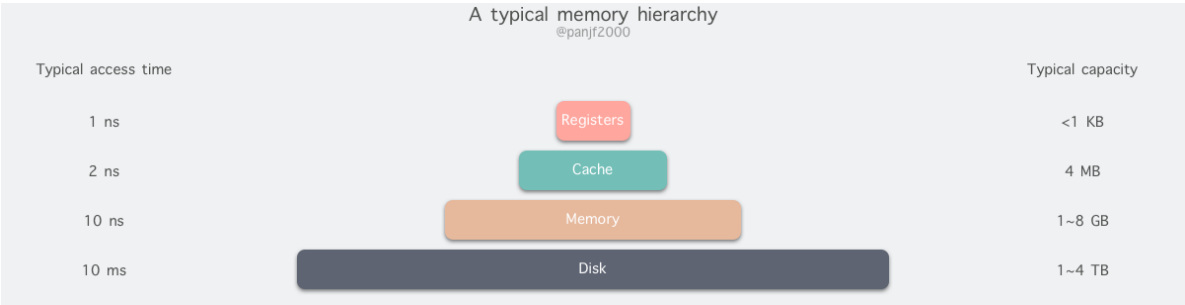
本文全面而深入地剖析了虚拟内存的工作原理，帮助读者快速而深刻地理解这个重要的概念。

计算机存储器

存储器是计算机的核心部件之一，在完全理想的状态下，存储器应该要同时具备以下三种特性：

1. 速度足够快：存储器的存取速度应当快于 CPU 执行一条指令，这样 CPU 的效率才不会受限于存储器
2. 容量足够大：容量能够存储计算机所需的全部数据
3. 价格足够便宜：价格低廉，所有类型的计算机都能配备

但是现实往往是残酷的，我们目前的计算机技术无法同时满足上述的三个条件，于是现代计算机的存储器设计采用了一种分层次的结构：



从顶至底，现代计算机里的存储器类型分别有：寄存器、高速缓存、主存和磁盘，这些存储器的速度逐级递减而容量逐级递增。存取速度最快的是寄存器，因为寄存器的制作材料和 CPU 是相同的，所以速度和 CPU 一样快，CPU 访问寄存器是没有时延的，然而因为价格昂贵，因此容量也极小，一般 32 位的 CPU 配备的寄存器容量是 32✕32 Bit，64 位的 CPU 则是 64✕64 Bit，不管是 32 位还是 64 位，寄存器容量都小于 1 KB，且寄存器也必须通过软件自行管理。

第二层是高速缓存，也即我们平时了解的 CPU 高速缓存

L1、L2、L3，一般 L1 是每个 CPU 独享，L3 是全部 CPU 共享，而 L2 则根据不同的架构设计会被设计成独享或者共享两种模式之一，比如 Intel 的多核芯片采用的是共享 L2 模式而 AMD 的多核芯片则采用的是独享 L2 模式。

第三层则是主存，也即主内存，通常称作随机访问存储器（Random Access Memory, RAM）。是与 CPU 直接交换数据的内部存储器。它可以随时读写（刷新时除外），而且速度很快，通常作为操作系统或其他正在运行中的程序的临时资料存储介质。

最后则是磁盘，磁盘和主存相比，每个二进制位的成本低了两个数量级，因此容量比之会大得多，动辄上 GB、TB，而缺点则是访问速度则比主存慢了大概三个数量级。机械硬盘速度慢主要是因为机械臂需要不断在金属盘片之间移动，等待磁盘扇区旋转至磁头之下，然后才能进行读写操作，因此效率很低。

主存

物理内存

我们平时一直提及的物理内存就是上文中对应的第三种计算机存储器，RAM 主存，它在计算机中以内存条的形式存在，嵌在主板的内存槽上，用来加载各式各样的程序与数据以供 CPU 直接运行和使用。

虚拟内存

在计算机领域有一句如同摩西十诫般神圣的哲言："**计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决**"，从内存管理、网络模型、并发调度甚至是硬件架构，都能看到这句哲言在闪烁着光芒，而虚拟内存则是这一哲言的完美实践之一。

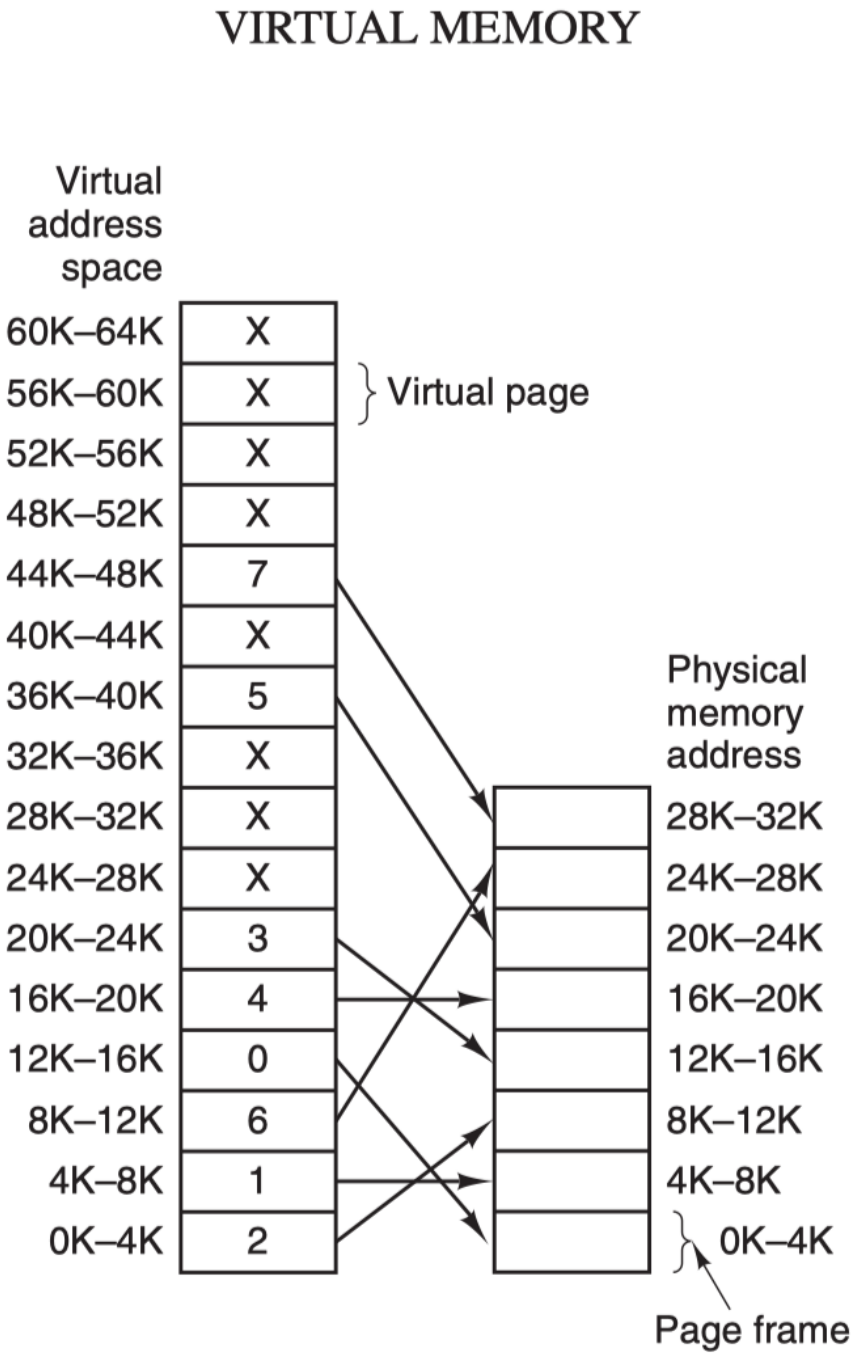
虚拟内存是现代计算机中的一个非常重要的存储器抽象，主要是用来解决应用程序日益增长的内存使用需求：现代物理内存的容量增长已经非常快速了，然而还是跟不上应用程序对主存需求的增长速度，对于应用程序来说内存还是可能会不够用，因此便需要一种方法来解决这两者之间的容量差矛盾。为了更高效地管理内存并尽可能消除程序错误，现代计算机系统对物理主存 RAM 进行抽象，实现了***虚拟内存 (Virtual Memory, VM)*** 技术。

虚拟内存

虚拟内存的核心原理是：为每个程序设置一段"连续"的虚拟地址空间，把这个地址空间分割成多个具有连续地址范围的页 (Page)，并把这些页和物理内存做映射，在程序运行期间动态映射到物理内存。当程序引用到一段在物理内存的地址空间时，由硬件立刻执行必要的映射；而当程序引用到一段不在物理内存中的地址空间

时，由操作系统负责将缺失的部分装入物理内存并重新执行失败的指令。

其实虚拟内存技术从某种角度来看的话，很像是糅合了基址寄存器和界限寄存器之后的新技术。它使得整个进程的地址空间可以通过较小的虚拟单元映射到物理内存，而不需要为程序的代码和数据地址进行重定位。



虚拟地址空间按照固定大小划分成被称为页（Page）的若干单元，物理内存中对应的则是页框（Page Frame）。这两者一般来说是一样的大小，如上图中的是 4KB，不过实际上计算机系统中一般是 512 字节到 1 GB，这就是虚拟内存的分页技术。因为是虚拟内存空间，每个进程分配的大小是 4GB (32 位架构)，而实际上当然不可能给所有在运行中的进程都分配 4GB 的物理内存，所以虚拟内存技术还需要利用到一种 交换（swapping）技术，也就是通常所说的页面置换算法，在进程运行期间只分配映射当前使用到的内存，暂时不使用的数据则写回磁盘作为副本保存，需要用的时候再读入内存，动态地在磁盘和内存之间交换数据。

页表

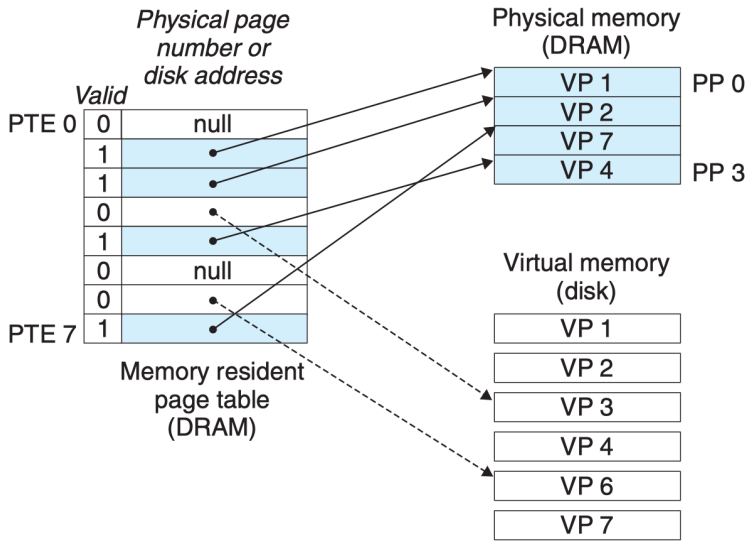
页表（Page Table），每次进行虚拟地址到物理地址的映射之时，都需要读取页表，从数学角度来说页表就是一个函数，入参是虚拟页号（Virtual Page Number，简称 VPN），输出是物理页框号（Physical Page Number，简称 PPN，也就是物理地址的基址）。

页表由多个页表项（Page Table Entry, 简称 PTE）组成，页表项的结构取决于机器架构，不过基本上都大同小异。一般来说页表项中都会存储物理页框号、修改位、访问位、保护位和 "在/不在" 位（有效位）等信息。

- 物理页框号：这是 PTE 中最重要的域值，毕竟页表存在的意义就是提供 VPN 到 PPN 的映射。
- 有效位：表示该页面当前是否存在于主存中，1 表示存在，0 表示缺失，当进程尝试访问一个有效位为 0 的页面时，就会引起一个缺页中断。
- 保护位：指示该页面所允许的访问类型，比如 0 表示可读写，1 表示只读。
- 修改位和访问位：为了记录页面使用情况而引入的，一般是页面置换算法会使用到。比如当一个内存页面被程序修改过之后，硬件会自动设置修改位，如果下次程序发生缺页中断需要运行页面置换算法把该页面调出以便为即将调入的页面腾出空间之时，就会先去访问修改位，从而得知该页面被修改过，也就是脏页 (Dirty Page)，则需要把最新的页面内容写回到磁盘保存，否则就表示内存和磁盘上的副本内容是同步的，无需写回磁盘；而访问位同样也是系统在程序访问页面时自动设置的，它也是页面置换算法会使用到的一个值，系统会根据页面是否正在被访问来觉得是否要淘汰掉这个页面，一般来说不再使用的页面更适合被淘汰掉。
- 高速缓存禁止位：用于禁止页面被放入 CPU 高速缓存，这个值主要适用于那些映射到寄存器等实时 I/O 设备而非普通主存的内存页面，这一类实时 I/O 设备需要拿到最新的数据，而 CPU 高速缓存中的数据可能是旧的拷

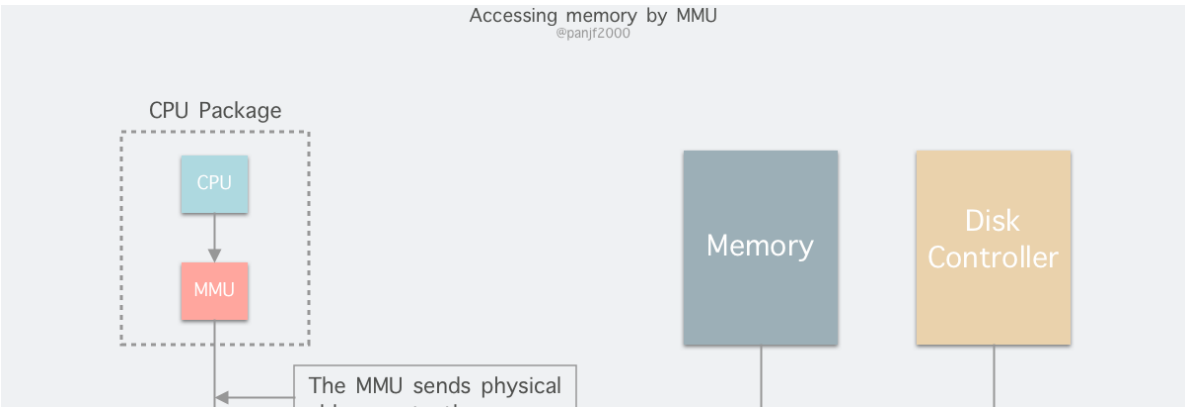
贝。

Figure 9.4
Page table.



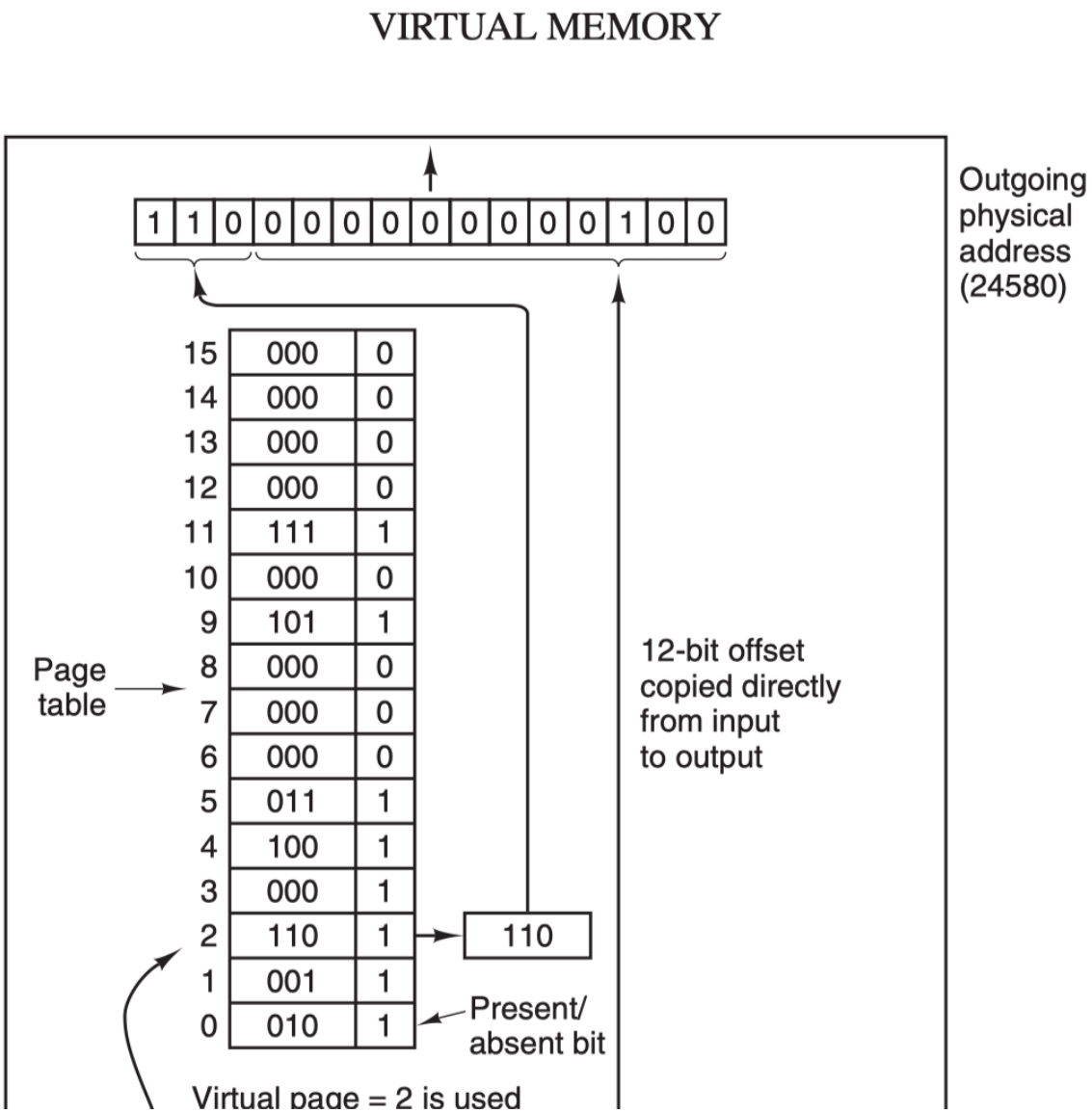
地址翻译

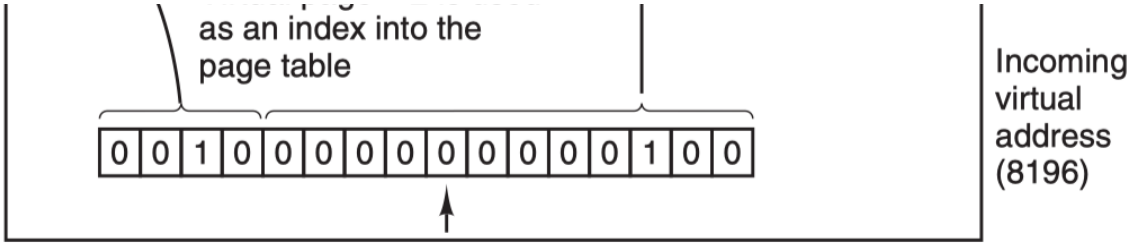
进程在运行期间产生的内存地址都是虚拟地址，如果计算机没有引入虚拟内存这种存储器抽象技术的话，则 CPU 会把这些地址直接发送到内存地址总线上，然后访问和虚拟地址相同值的物理地址；如果使用虚拟内存技术的话，CPU 则是把这些虚拟地址通过地址总线送到内存管理单元（Memory Management Unit，简称 MMU），MMU 将虚拟地址翻译成物理地址之后再通过内存总线去访问物理内存：



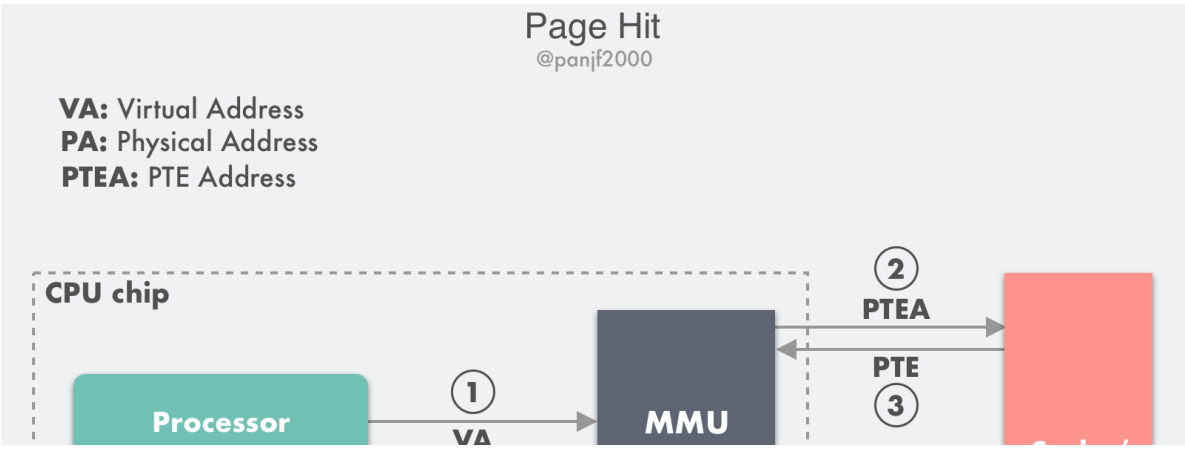
虚拟地址（比如 16 位地址 8196=0010 000000000100）分为两部分：虚拟页号（Virtual Page Number，简称 VPN，这里是高 4 位部分）和偏移量（Virtual Page Offset，简称 VPO，这里是低 12 位部分），虚拟地址转换成物理地址是通过页表（page table）来实现的。

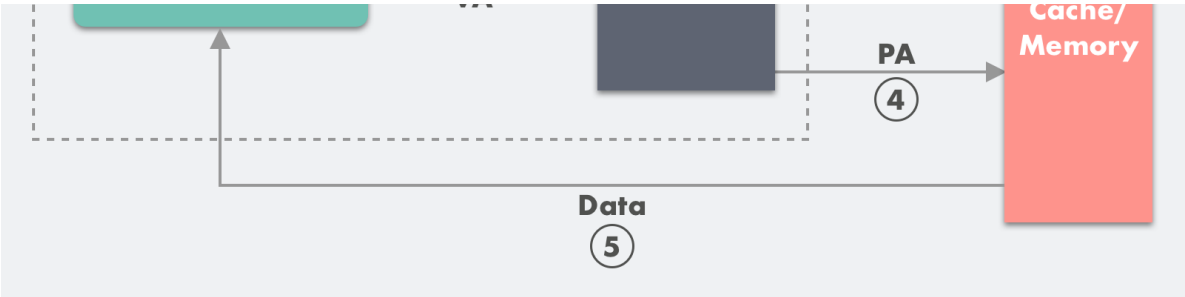
这里我们基于一个例子来分析当页面命中时，计算机各个硬件是如何交互的：





- **第 1 步**：处理器生成一个虚拟地址 VA，通过总线发送到 MMU；
- **第 2 步**：MMU 通过虚拟页号得到页表项的地址 PTEA，通过内存总线从 CPU 高速缓存/主存读取这个页表项 PTE；
- **第 3 步**：CPU 高速缓存或者主存通过内存总线向 MMU 返回页表项 PTE；
- **第 4 步**：MMU 先把页表项中的物理页框号 PPN 复制到寄存器的高三位中，接着把 12 位的偏移量 VPO 复制到寄存器的末 12 位构成 15 位的物理地址，即可以把该寄存器存储的物理内存地址 PA 发送到内存总线，访问高速缓存/主存；
- **第 5 步**：CPU 高速缓存/主存返回该物理地址对应的数据给处理器。





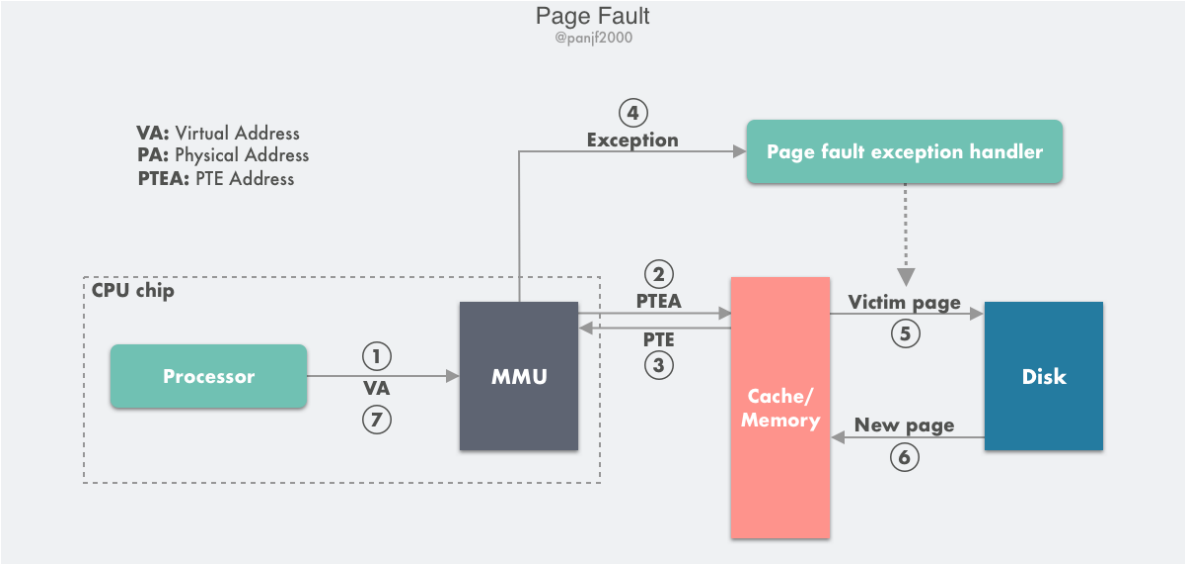
在 MMU 进行地址转换时，如果页表项的有效位是 0，则表示该页面并没有映射到真实的物理页框号 PPN，则会引发一个**缺页中断**，CPU 陷入操作系统内核，接着操作系统就会通过页面置换算法选择一个页面将其换出 (swap)，以便为即将调入的新页面腾出位置，如果要换出的页面的页表项里的修改位已经被设置过，也就是被更新过，则这是一个脏页 (Dirty Page)，需要写回磁盘更新该页面在磁盘上的副本，如果该页面是"干净"的，也就是没有被修改过，则直接用调入的新页面覆盖掉被换出的旧页面即可。

缺页中断的具体流程如下：

- **第 1 步到第 3 步**：和前面的页面命中的前 3 步是一致的；
- **第 4 步**：检查返回的页表项 PTE 发现其有效位是 0，则 MMU 触发一次缺页中断异常，然后 CPU 转入到操作系统内核中的缺页中断处理器；
- **第 5 步**：缺页中断处理程序检查所需的虚拟地址是否合法，确认合法后系统则检查是否有空闲物理页框号 PPN 可以映射给该缺失的虚拟页面，如果没有空闲页框，则

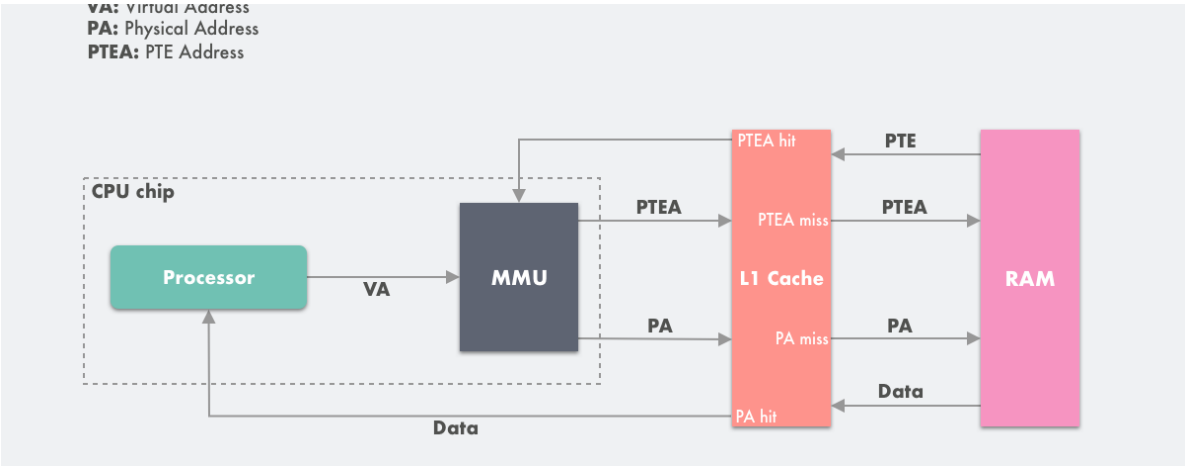
执行页面置换算法寻找一个现有的虚拟页面淘汰，如果该页面已经被修改过，则写回磁盘，更新该页面在磁盘上的副本；

- **第 6 步**：缺页中断处理程序从磁盘调入新的页面到内存，更新页表项 PTE；
- **第 7 步**：缺页中断程序返回到原先的进程，重新执行引起缺页中断的指令，CPU 将引起缺页中断的虚拟地址重新发送给 MMU，此时该虚拟地址已经有了映射的物理页框号 PPN，因此会按照前面『Page Hit』的流程走一遍，最后主存把请求的数据返回给处理器。



虚拟内存和高速缓存

前面在分析虚拟内存的工作原理之时，谈到页表的存储位置，为了简化处理，都是默认把主存和高速缓存放在一起，而实际上更详细的流程应该是如下的原理图：



如果一台计算机同时配备了虚拟内存技术和 CPU 高速缓存，那么 MMU 每次都会优先尝试到高速缓存中进行寻址，如果缓存命中则会直接返回，只有当缓存不命中之后才去主存寻址。

通常来说，大多数系统都会选择利用物理内存地址去访问高速缓存，因为高速缓存相比于主存要小得多，所以使用物理寻址也不会太复杂；另外也因为高速缓存容量很小，所以系统需要尽量在多个进程之间共享数据块，而使用物理地址能够使得多进程同时在高速缓存中存储数据块以及共享来自相同虚拟内存页的数据块变得更加直观。

加速翻译&优化页表

经过前面的剖析，相信读者们已经了解了虚拟内存及其分页&地址翻译的基础和原理。现在我们可以引入虚拟内存中两个核心的需求，或者说瓶颈：

- 虚拟地址到物理地址的映射过程必须要非常快，地址翻

译如何加速。

- 虚拟地址范围的增大必然会导致页表的膨胀，形成大页表。

这两个因素决定了虚拟内存这项技术能不能真正地广泛应用到计算机中，如何解决这两个问题呢？

正如文章开头所说："**计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决**"。因此，虽然虚拟内存本身就已经是一个中间层了，但是中间层里的问题同样可以通过再引入一个中间层来解决。

加速地址翻译过程的方案目前是通过引入页表缓存模块 -- TLB，而大页表则是通过实现多级页表或倒排页表来解决。

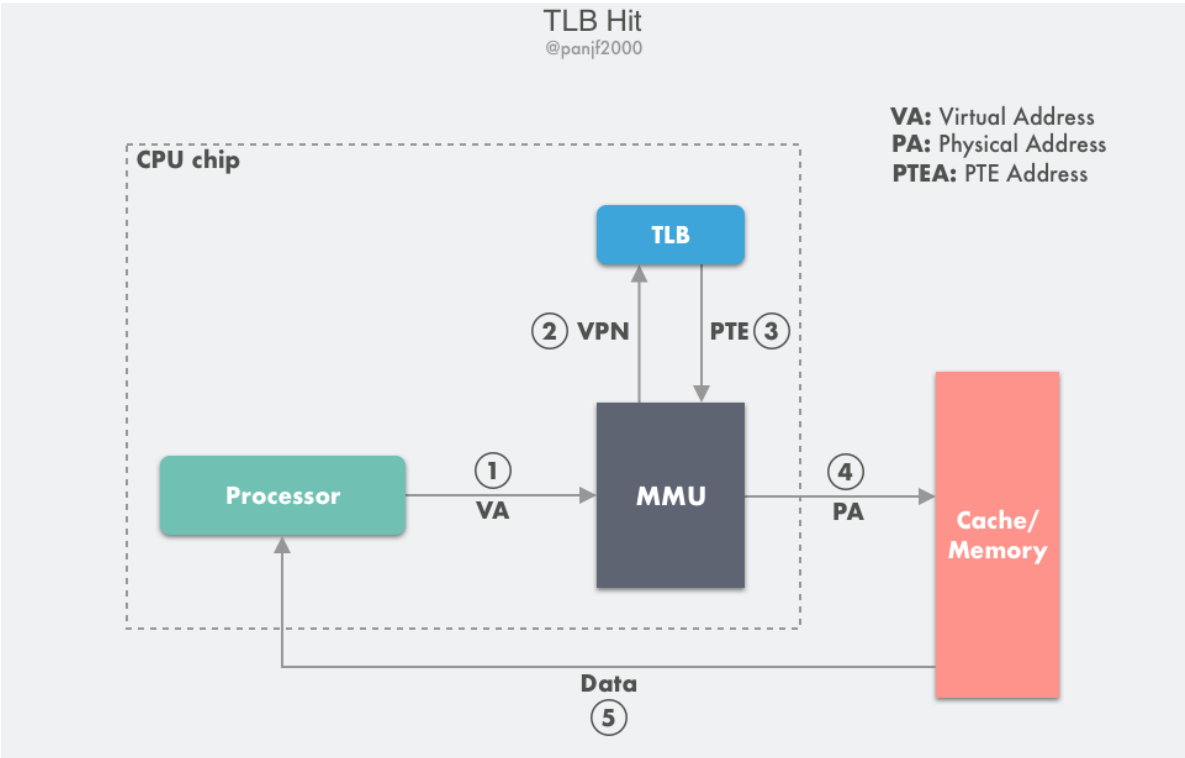
TLB 加速

翻译后备缓冲器（Translation Lookaside Buffer，TLB），也叫快表，是用来加速虚拟地址翻译的，因为虚拟内存的分页机制，页表一般是保存在内存中的一块固定的存储区，而 MMU 每次翻译虚拟地址的时候都需要从页表中匹配一个对应的 PTE，导致进程通过 MMU 访问指定内存数据的时候比没有分页机制的系统多了一次内存访问，一般会多耗费几十到几百个 CPU 时钟周期，性能至少下降一半，如果 PTE 碰巧缓存在 CPU L1 高速缓存中，则开销可以降低到一两个周期，

但是我们不能寄希望于每次要匹配的 PTE 都刚好在 L1 中，因此需要引入加速机制，即 TLB 快表。

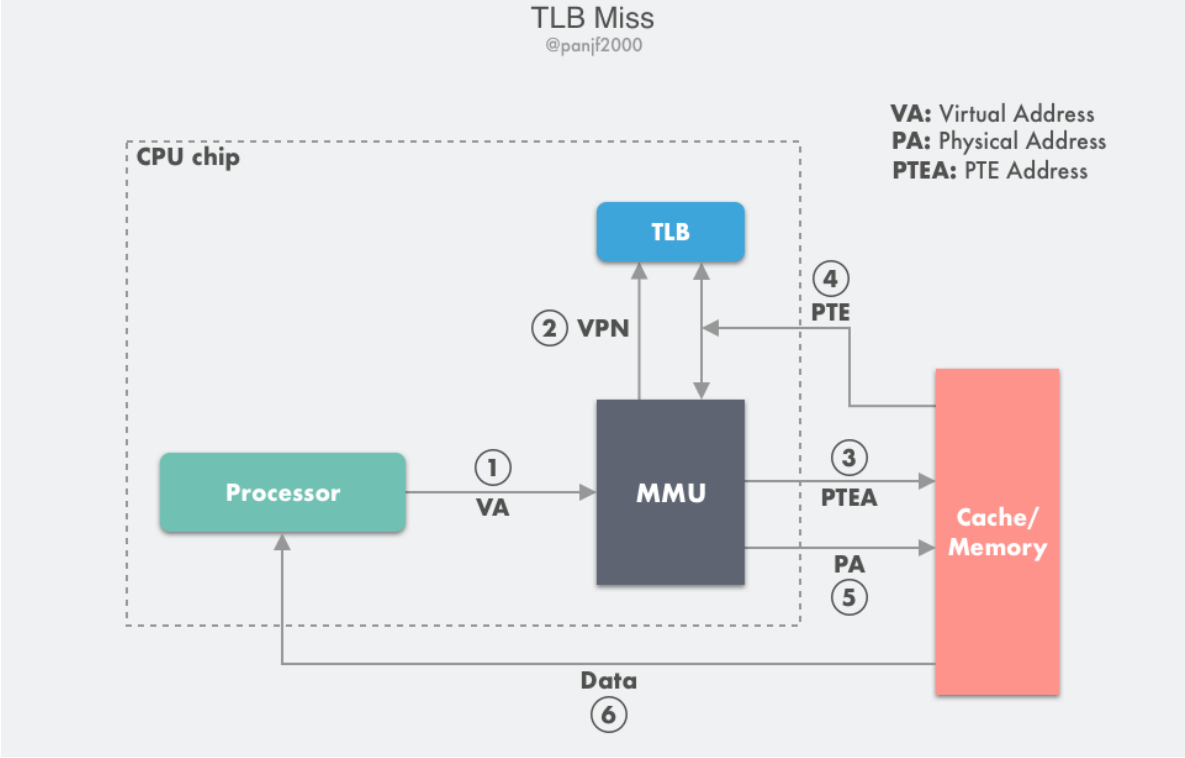
TLB 可以简单地理解成页表的高速缓存，保存了最高频被访问的页表项 PTE。由于 TLB 一般是硬件实现的，因此速度极快，MMU 收到虚拟地址时一般会先通过硬件 TLB 并行地在页表中匹配对应的 PTE，若命中且该 PTE 的访问操作不违反保护位（比如尝试写一个只读的内存地址），则直接从 TLB 取出对应的物理页框号 PPN 返回，若不命中则会穿透到主存页表里查询，并且会在查询到最新页表项之后存入 TLB，以备下次缓存命中，如果 TLB 当前的存储空间不足则会替换掉现有的其中一个 PTE。

下面来具体分析一下 TLB 命中和不命中。



TLB 命中：

- **第 1 步**：CPU 产生一个虚拟地址 VA；
- **第 2 步和第 3 步**：MMU 从 TLB 中取出对应的 PTE；
- **第 4 步**：MMU 将这个虚拟地址 VA 翻译成一个真实的物理地址 PA，通过地址总线发送到高速缓存/主存中去；
- **第 5 步**：高速缓存/主存将物理地址 PA 上的数据返回给 CPU。



TLB 不命中：

- **第 1 步**：CPU 产生一个虚拟地址 VA；
- **第 2 步至第 4 步**：查询 TLB 失败，走正常的主存页表查询流程拿到 PTE，然后把它放入 TLB 缓存，以备下次查询，如果 TLB 此时的存储空间不足，则这个操作会汰换掉 TLB 中另一个已存在的 PTE；

- **第 5 步**：MMU 将这个虚拟地址 VA 翻译成一个真实的物理地址 PA，通过地址总线发送到高速缓存/主存中去；
- **第 6 步**：高速缓存/主存将物理地址 PA 上的数据返回给 CPU。

多级页表

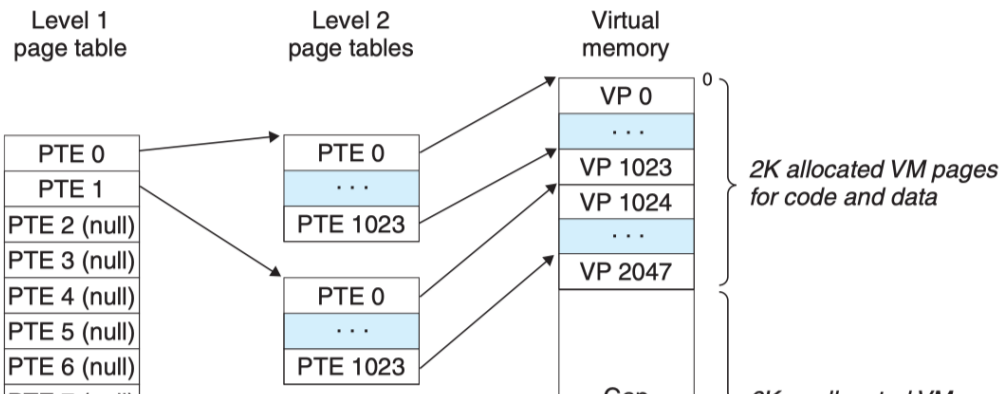
TLB 的引入可以一定程度上解决虚拟地址到物理地址翻译的开销问题，接下来还需要解决另一个问题：大页表。

理论上一台 32 位的计算机的寻址空间是 4GB，也就是说每一个运行在该计算机上的进程理论上的虚拟寻址范围是 4GB。到目前为止，我们一直在讨论的都是单页表的情形，如果每一个进程都把理论上可用的内存页都装载进一个页表里，但是实际上进程会真正使用到的内存其实可能只有很小的一部分，而我们也知道页表也是保存在计算机主存中的，那么势必会造成大量的内存浪费，甚至有可能导致计算机物理内存不足从而无法并行地运行更多进程。

这个问题一般通过**多级页表**（Multi-Level Page Tables）来解决，通过把一个大页表进行拆分，形成多级的页表，我们具体来看一个二级页表应该如何设计：假定一个虚拟地址是 32 位，由 10 位的一级页表索引、10 位的二级页表索引以及 12 位的地址偏移量，则 PTE 是 4

字节，页面 page 大小是 2^12 = 4KB，总共需要 2^20 个 PTE，一级页表中的每个 PTE 负责映射虚拟地址空间中的一个 4MB 的 chunk，每一个 chunk 都由 1024 个连续的页面 Page 组成，如果寻址空间是 4GB，那么一共只需要 1024 个 PTE 就足够覆盖整个进程地址空间。二级页表中的每一个 PTE 都负责映射到一个 4KB 的虚拟内存页面，和单页表的原理是一样的。

多级页表的关键在于，我们并不需要为一级页表中的每一个 PTE 都分配一个二级页表，而只需要为进程当前使用到的地址做相应的分配和映射。因此，对于大部分进程来说，它们的一级页表中有大量空置的 PTE，那么这部分 PTE 对应的二级页表也将无需存在，这是一个相当可观的内存节约，事实上对于一个典型的程序来说，理论上的 4GB 可用虚拟内存地址空间绝大部分都会处于这样一种未分配的状态；更进一步，在程序运行过程中，只需要把一级页表放在主存中，虚拟内存系统可以在实际需要的时候才去创建、调入和调出二级页表，这样就可以确保只有那些最频繁被使用的二级页表才会常驻在主存中，此举亦极大地缓解了主存的压力。



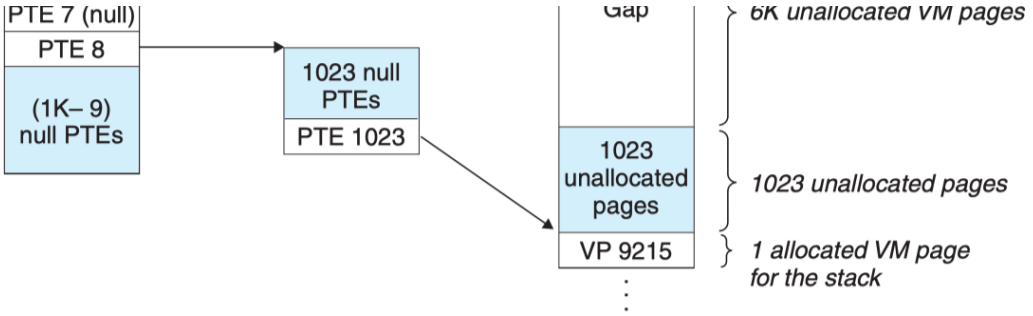
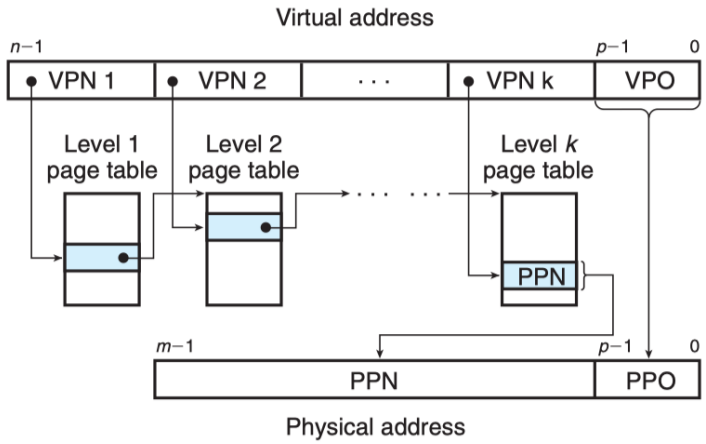


Figure 9.17 **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

多级页表的层级深度可以按照需求不断扩充，一般来说，级数越多，灵活性越高。

Figure 9.18
Address translation with a *k*-level page table.



比如有个一个 k 级页表，虚拟地址由 k 个 VPN 和 1 个 VPO 组成，每一个 VPN i 都是一个到第 i 级页表的索引，其中 1 <= i <= k。第 j 级页表中的每一个 PTE（1 <= j <= k-1）都指向第 j+1 级页表的基址。第 k 级页表中的每一个 PTE 都包含一个物理地址的页框号 PPN，或者一个磁盘块的地址（该内存页已经被页面置换算法换出到磁盘中）。MMU 每次都需要访问 k 个 PTE 才能找到物理页框号 PPN 然后加上虚拟地址中的偏移量 VPO 从而生成一个物理地址。这里读者可能会对 MMU 每次都访问 k 个 PTE 表示性能上的担忧，此时就是

TLB 出场的时候了，计算机正是通过把每一级页表中的 PTE 缓存在 TLB 中从而让多级页表的性能不至于落后单页表太多。

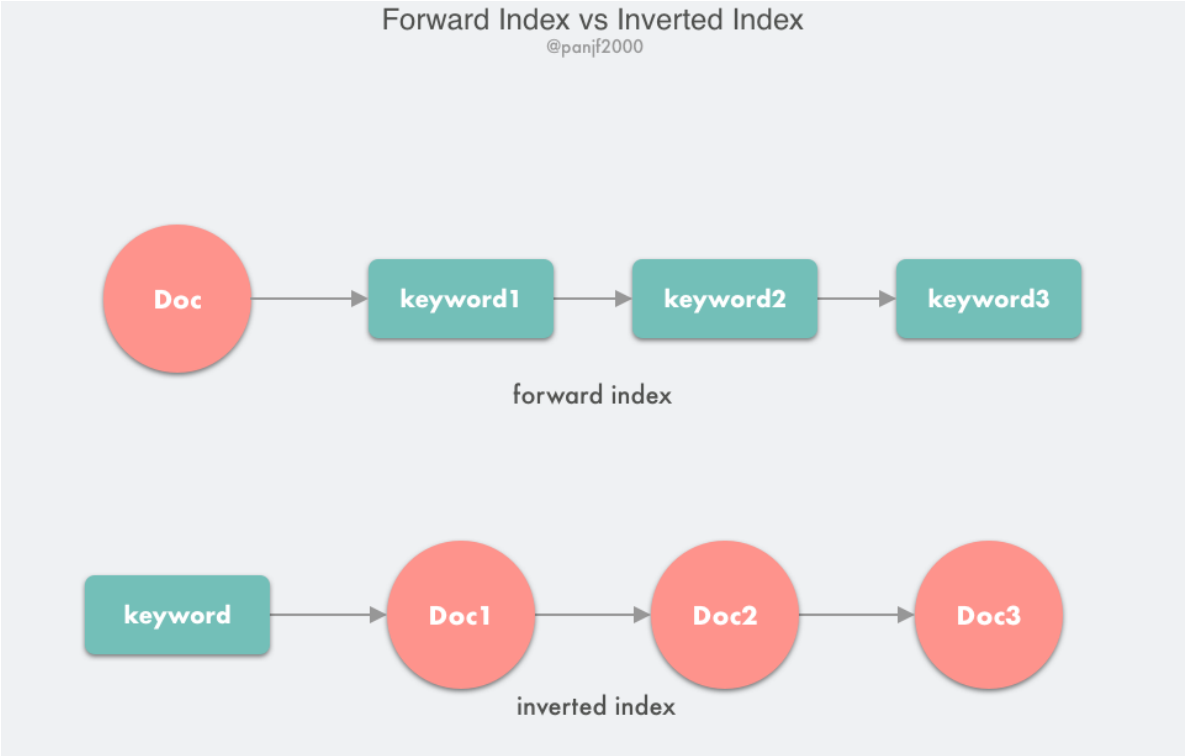
倒排页表

另一种针对页式虚拟内存管理大页表问题的解决方案是**倒排页表**（Inverted Page Table，简称 IPT）。倒排页表的原理和搜索引擎的倒排索引相似，都是通过反转映射过程来实现。

在搜索引擎中，有两个概念：文档 doc 和 关键词 keyword，我们的需求是通过 keyword 快速找到对应的 doc 列表，如果搜索引擎的存储结构是正向索引，也即是通过 doc 映射到其中包含的所有 keyword 列表，那么我们要找到某一个指定的 keyword 所对应的 doc 列表，那么便需要扫描索引库中的所有 doc，找到包含该 keyword 的 doc，再根据打分模型进行打分，排序后返回，这种设计无疑是低效的；所以我们需要反转一下正向索引从而得到倒排索引，也即通过 keyword 映射到所有包含它的 doc 列表，这样当我们查询包含某个指定 keyword 的 doc 列表时，只需要利用倒排索引就可以快速定位到对应的结果，然后还是根据打分模型进行排序返回。

上面的描述只是搜索引擎倒排索引的简化原理，实际的

倒排索引设计是要复杂很多的，有兴趣的读者可以自行查找资料学习，这里就不再展开。



回到虚拟内存的倒排页表，它正是采用了和倒排索引类似的思想，反转了映射过程：前面我们学习到的页表设计都是以虚拟地址页号 VPN 作为页表项 PTE 索引，映射到物理页框号 PPN，而在倒排页表中则是以 PPN 作为 PTE 索引，映射到 (进程号，虚拟页号 VPN)。

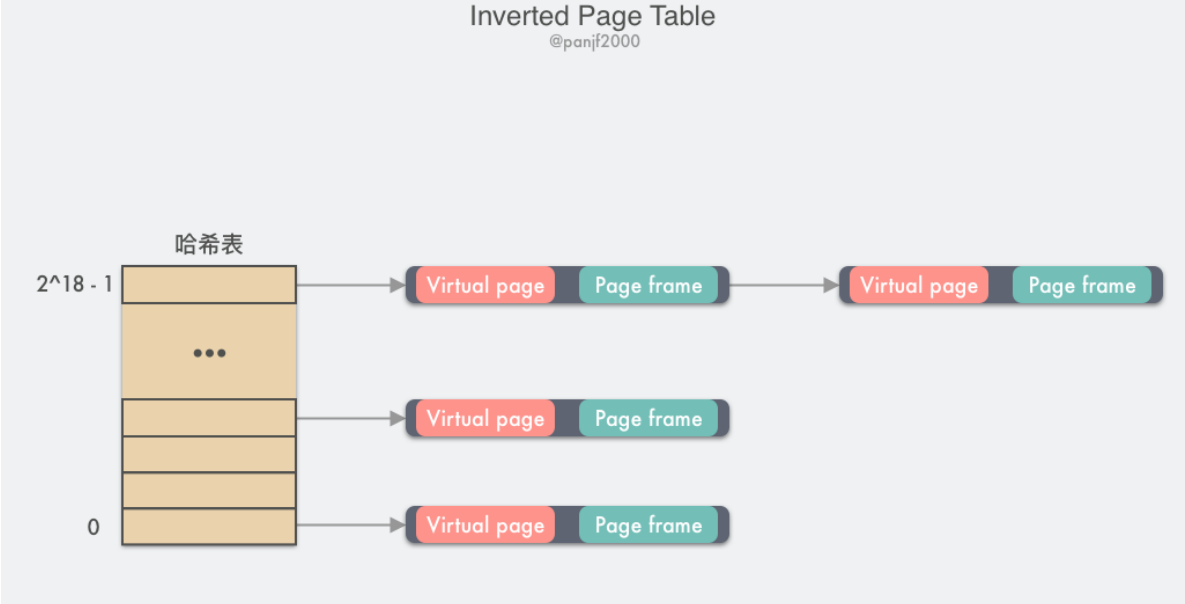
倒排页表在寻址空间更大的 CPU 架构下尤其高效，或者应该说更适合那些『虚拟内存空间/物理内存空间』比例非常大的场景，因为这种设计是以实际物理内存页框作为 PTE 索引，而不是以远超物理内存的虚拟内存作为索引。例如，以 64 位架构为例，如果是单页表结构，还是用 12 位作为页面地址偏移量，也就是 4KB 的内存页大小，那么以最理论化的方式来计算，则需要 2^52 个

PTE，每个 PTE 占 8 个字节，那么整个页表需要 32PB 的内存空间，这完全是不可接受的，而如果采用倒排页表，假定使用 4GB 的 RAM，则只需要 2^20 个 PTE，极大减少内存使用量。

倒排页表虽然在节省内存空间方面效果显著，但同时却引入了另一个重大的缺陷：地址翻译过程变得更加低效。我们都清楚 MMU 的工作就是要把虚拟内存地址翻译成物理内存地址，现在索引结构变了，物理页框号 PPN 作为索引，从原来的 VPN --> PPN 变成了 PPN --> VPN，那么当进程尝试访问一个虚拟内存地址之时，CPU 在通过地址总线把 VPN 发送到 MMU 之后，基于倒排页表的设计，MMU 并不知道这个 VPN 对应的是不是一个缺页，所以不得不扫描整个倒排页表来找到该 VPN，而最要命的是就算是一个非缺页的 VPN，每次内存访问还是需要执行这个全表扫描操作，假设是前面提到的 4GB RAM 的例子，那么相当于每次都要扫描 2^20 个 PTE，相当低效。

这时候又是我们的老朋友 -- TLB 出场的时候了，我们只需要把高频使用的页面缓存在 TLB 中，借助于硬件，在 TLB 缓存命中的情况下虚拟内存地址的翻译过程就可以像普通页表那样快速，然而当 TLB 失效的时候，则还是需要通过软件的方式去扫描整个倒排页表，线性扫描的方式非常低效，因此一般倒排页表会基于哈希表来实现，假设有 1G 的物理内存，那么这里就一共有 2^18 个

4KB 大小的页框，建立一张以 PPN 作为 key 的哈希表，则可以划分成 2^18 个 4KB 大小的页框，假设 0 作为 PPN 的起点，则 [0, 2^18 - 1] 就是 PPN 的取值范围，以此作为 Hash map 的 key，然后实现一个哈希函数，使用 VPN 作为入参，使得哈希函数最后输出的哈希值落在 [0, 2^18 - 1] 区间内，每一个 key 值对应的 value 中存储的是 (VPN, PNN)，那么所有具有相同哈希值的 VPN 会被链接在一起形成一个冲突链，如果我们把哈希表的槽数设置成跟物理页框数量一致的话，那么这个倒排哈希表中的冲突链的平均长度将会是 1 个 PTE，可以大大提高查询速度。当 VPN 通过倒排页表匹配到 PPN 之后，这个 (VPN, PPN) 映射关系就会马上被缓存进 TLB，以加速下次虚拟地址翻译。



倒排页表在 64 位架构的计算机中很常见，因为在 64 位架构下，基于分页的虚拟内存中即便把页面 Page 的大小从一般的 4KB 提升至 4MB，依然需要一个拥有 2^42

个 PTE 的巨型页表放在主存中（理论上，实际上不会这么实现），极大地消耗内存。

总结

现在让我们来回顾一下本文的核心内容：虚拟内存是存在于计算机 CPU 和物理内存之间一个中间层，主要作用是高效管理内存并减少内存出错。虚拟内存的几个核心概念有：

1. **页表**：从数学角度来说页表就是一个函数，入参是虚拟页号 VPN，输出是物理页框号 PPN，也就是物理地址的基址。页表由页表项组成，页表项中保存了所有用来进行地址翻译所需的信息，页表是虚拟内存得以正常运作的基础，每一个虚拟地址要翻译成物理地址都需要借助它来完成。
2. **TLB**：计算机硬件，主要用来解决引入虚拟内存之后寻址的性能问题，加速地址翻译。如果没有 TLB 来解决虚拟内存的性能问题，那么虚拟内存将只可能是一个学术上的理论而无法真正广泛地应用在计算机中。
3. **多级页表和倒排页表**：用来解决虚拟地址空间爆炸性膨胀而导致的大页表问题，多级页表通过将单页表进行分拆并按需分配虚拟内存页而倒排页表则是通过反转映射关系来实现节省内存的效果。

最后，虚拟内存技术中还需要涉及到操作系统的页面置

换机制，由于页面置换机制也是一个较为庞杂和复杂的概念，本文便不再继续剖析这一部分的原理，我们在以后的文章中再单独拿来讲解。

参考&延伸阅读

本文的主要参考资料是《现代操作系统》和《深入理解计算机系统》这两本书的英文原版，如果读者还想更加深入地学习虚拟内存，可以深入阅读这两本书并且搜寻其他的论文资料进行学习。

Q.E.D.