虚拟内存的那点事儿

2017-10-29 | 计算机 |

概述

我们都知道一个进程是与其他进程共享CPU和内存资源的。正因如此，操作系统需要有一套完善的内存管理机制才能防止进程之间内存泄漏的问题。

为了更加有效地管理内存并减少出错，现代操作系统提供了一种对主存的抽象概念，即是虚拟内存（Virtual Memory）。**虚拟内存为每个进程提供了一个一致的、私有的地址空间，它让每个进程产生了一种自己在独享主存的错觉（每个进程拥有一片连续完整的内存空间）**。

理解不深刻的人会认为虚拟内存只是“使用硬盘空间来扩展内存“的技术，这是不对的。**虚拟内存的重要意义是它定义了一个连续的虚拟地址空间**，使得程序的编写难度降低。并且，**把内存扩展到硬盘空间只是使用虚拟内存的必然结果，虚拟内存空间会存在硬盘中，并且会被内存缓存（按需），有的操作系统还会在内存不够的情况下，将某一进程的内存全部放入硬盘空间中，并在切换到该进程时再从硬盘读取**（这也是为什么Windows会经常假死的原因…）。

虚拟内存主要提供了如下三个重要的能力：

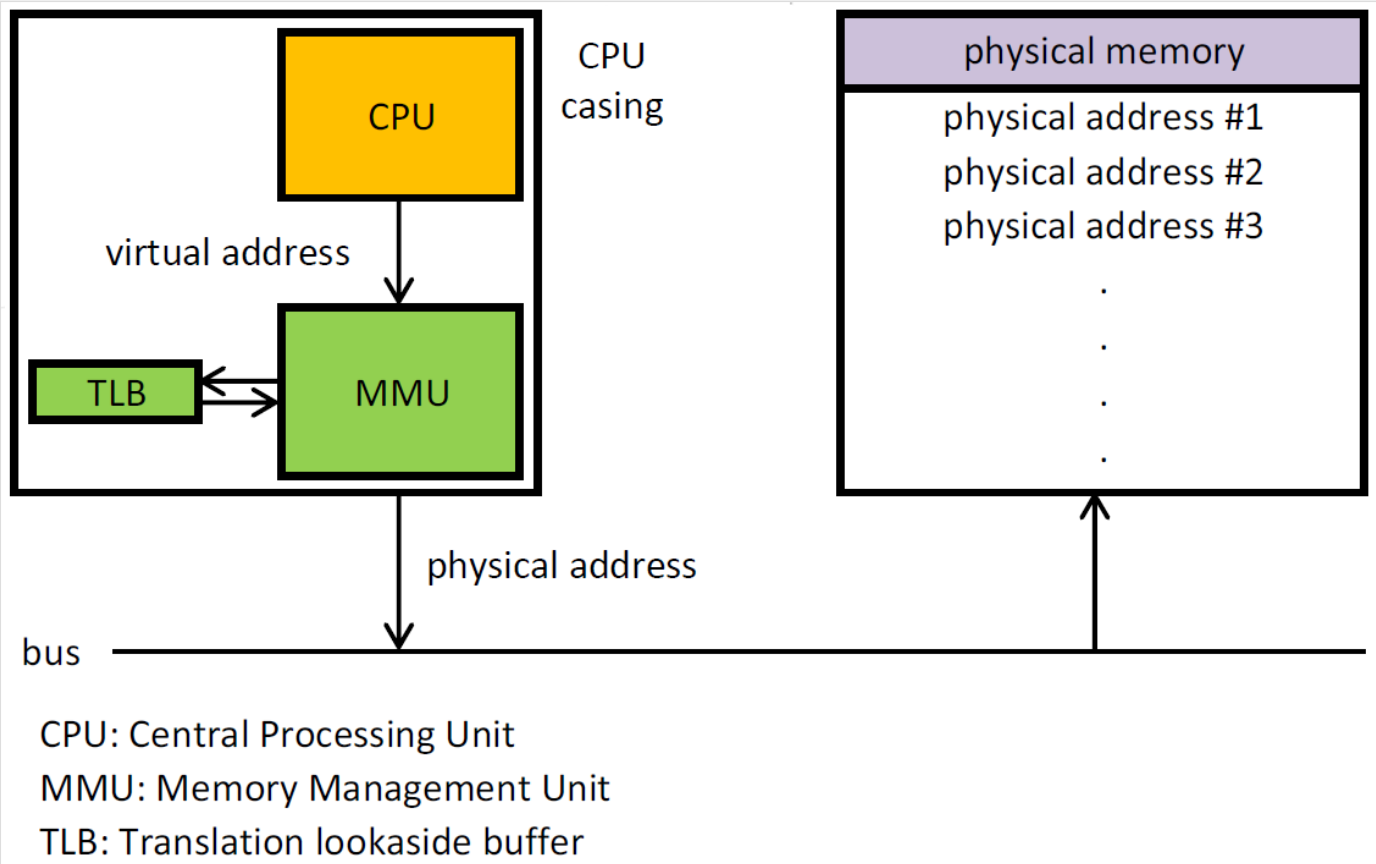
- 它把主存看作为一个存储在硬盘上的虚拟地址空间的高速缓存，并且只在主存中缓存活动区域（按需缓存）。
- 它为每个进程提供了一个一致的地址空间，从而降低了程序员对内存管理的复杂性。
- 它还保护了每个进程的地址空间不会被其他进程破坏。

介绍了虚拟内存的基本概念之后，接下来的内容将会从虚拟内存在硬件中如何运作逐渐过渡到虚拟内存在操作系统（Linux）中的实现。

CPU寻址

内存通常被组织为一个由M个连续的字节大小的单元组成的数组，每个字节都有一个唯一的物理地址（Physical Address PA），作为到数组的索引。CPU访问内存最简单直接的方法就是使用物理地址，这种寻址方式被称为物理寻址。

现代处理器使用的是一种称为虚拟寻址（Virtual Addressing）的寻址方式。**使用虚拟寻址，CPU需要将虚拟地址翻译成物理地址，这样才能访问到真实的物理内存。**



虚拟寻址需要硬件与操作系统之间互相合作。**CPU中含有一个被称为内存管理单元（Memory Management Unit, MMU）的硬件，它的功能是将虚拟地址转换为物理地址。MMU需要借助存放在内存中的页表来动态翻译虚拟地址，该页表由操作系统管理。**

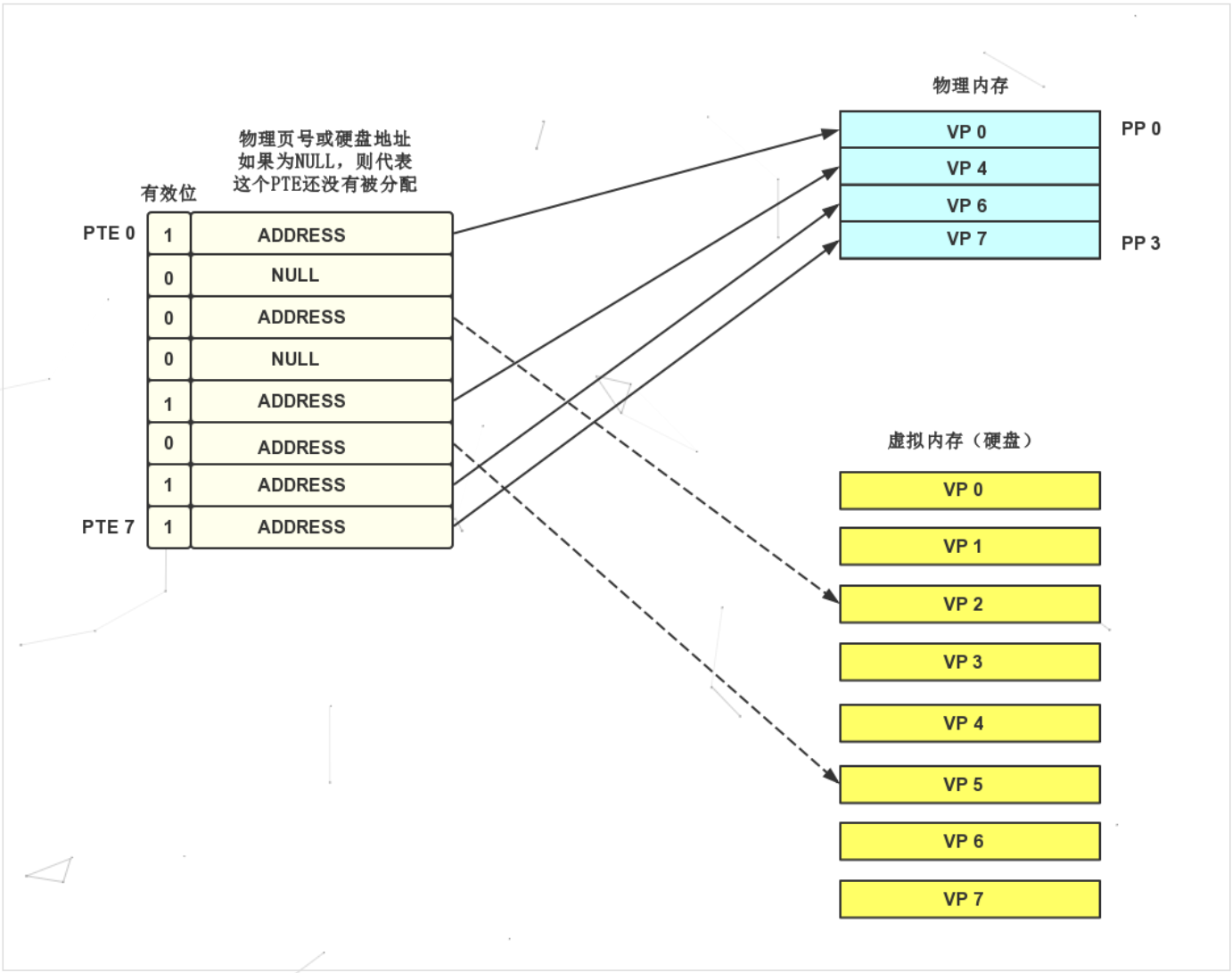
页表

虚拟内存空间被组织为一个存放在硬盘上的M个连续的字节大小的单元组成的数组，每个字节都有一个唯一的虚拟地址，作为到数组的索引（这点其实与物理内存是一样的）。

操作系统通过将虚拟内存分割为大小固定的块来作为硬盘和内存之间的传输单位，这个块被称为虚拟页（Virtual Page, VP），每个虚拟页的大小为 P=2^p 字节。物理内存也会按照这种方法分割为物理页（Physical Page, PP），大小也为 P 字节。

CPU在获得虚拟地址之后，需要通过MMU将虚拟地址翻译为物理地址。而在翻译的过程中还需要借助页表，所谓**页表就是一个存放在物理内存中的数据结构，它记录了虚拟页与物理页的映射关系。**

页表是一个元素为页表条目（Page Table Entry, PTE）的集合，每个虚拟页在页表中一个固定偏移量的位置上都有一个PTE。下面是PTE仅含有一个有效位标记的页表结构，该有效位代表这个虚拟页是否被缓存在物理内存中。

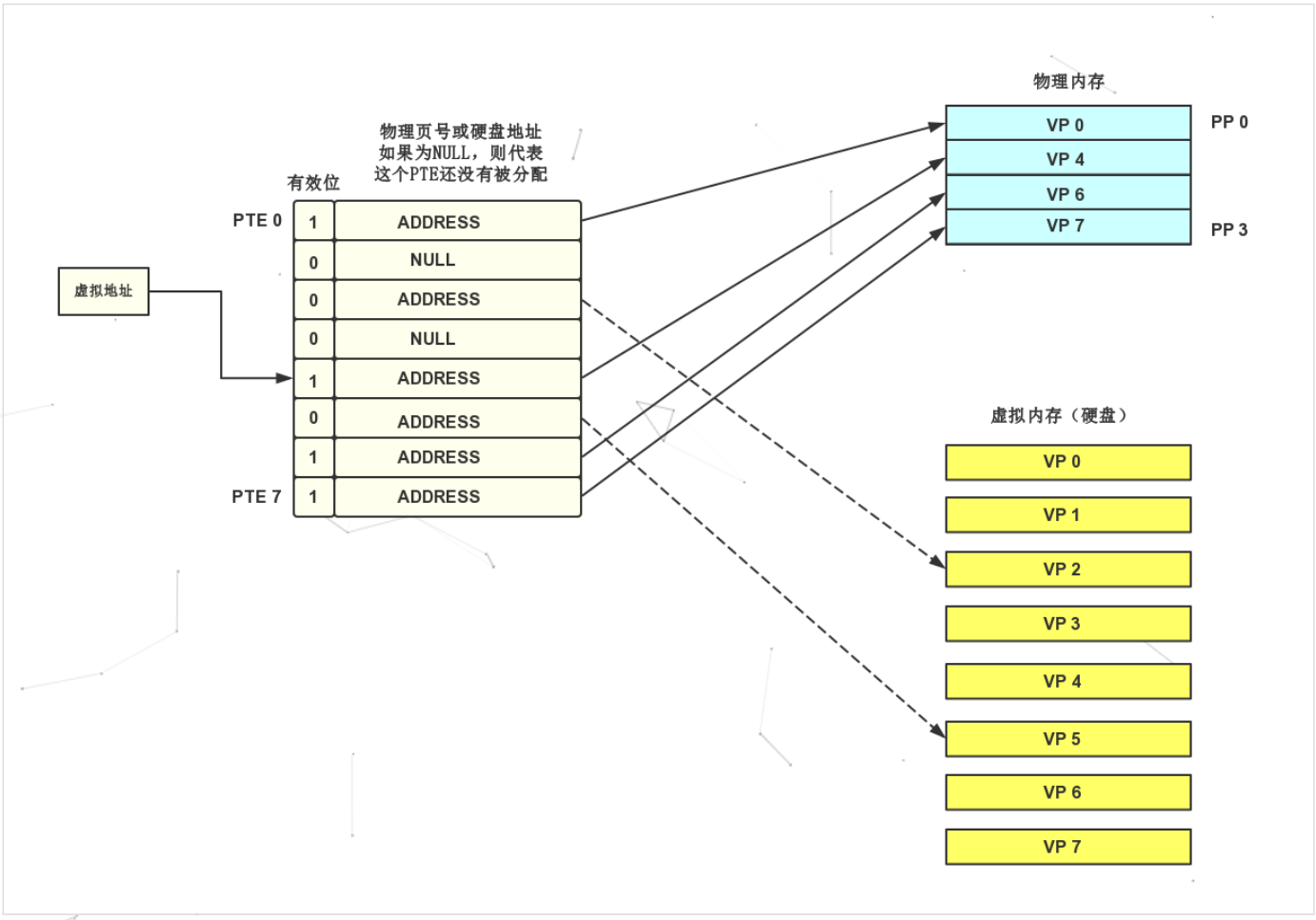


虚拟页 VP 0 、 VP 4 、 VP 6 、 VP 7 被缓存在物理内存中，虚拟页 VP 2 和 VP 5 被分配在页表中，但并没有缓存在物理内存，虚拟页 VP 1 和 VP 3 还没有被分配。

在进行动态内存分配时，例如 malloc() 函数或者其他高级语言中的 new 关键字，操作系统会在硬盘中创建或申请一段虚拟内存空间，并更新到页表（分配一个PTE，使该PTE指向硬盘上这个新创建的虚拟页）。

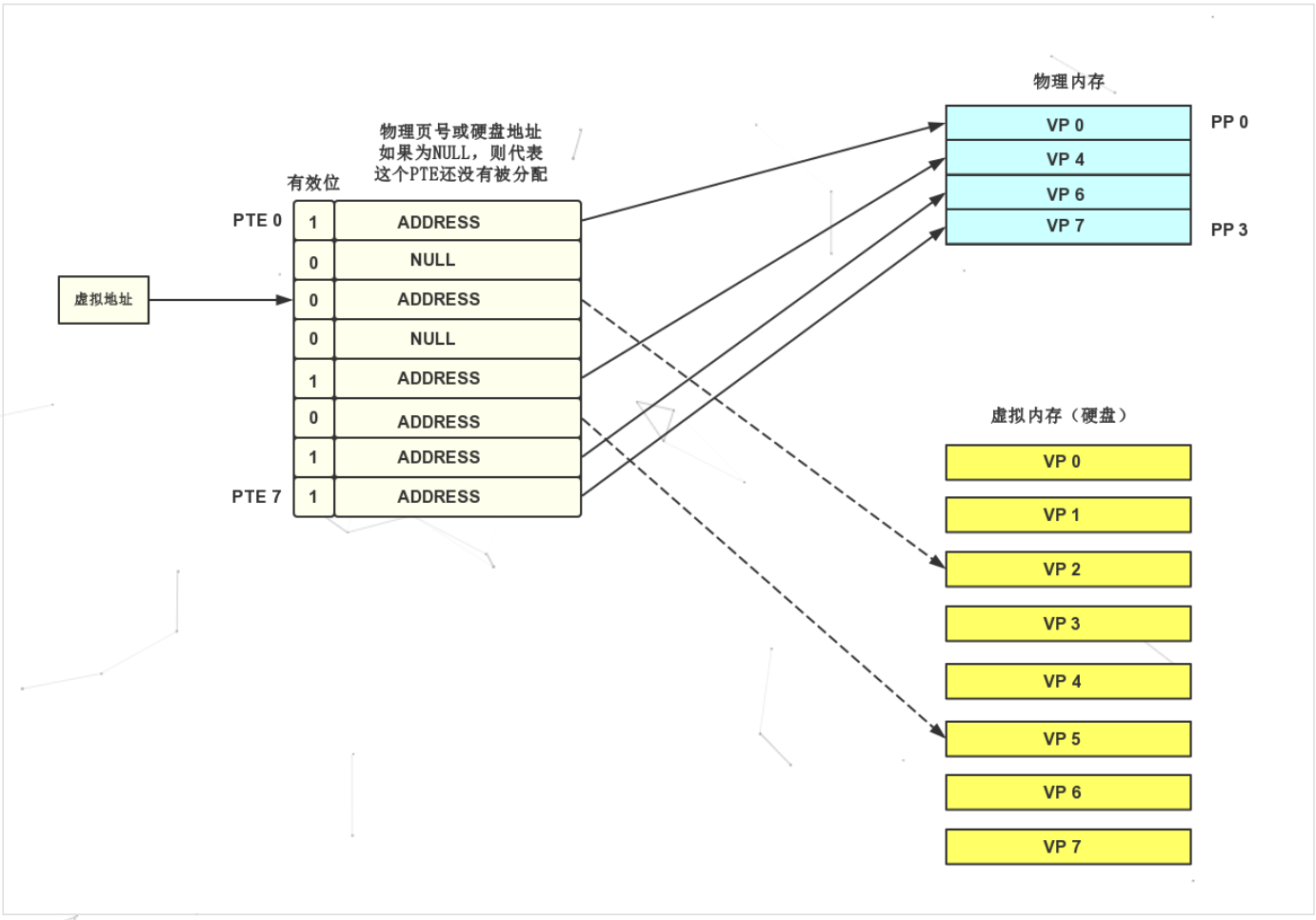
由于CPU每次进行地址翻译的时候都需要经过PTE，所以如果想控制内存系统的访问，可以在PTE上添加一些额外的许可位（例如读写权限、内核权限等），这样只要有指令违反了这些许可条件，CPU就会触发一个一般保护故障，将控制权传递给内核中的异常处理程序。一般这种异常被称为“段错误（Segmentation Fault）”。

页命中



如上图所示，MMU根据虚拟地址在页表中寻址到了 PTE 4，该PTE的有效位为1，代表该虚拟页已经被缓存在物理内存中了，最终MMU得到了PTE中的物理内存地址（指向 PP 1）。

缺页



如上图所示，MMU根据虚拟地址在页表中寻址到了 PTE 2，该PTE的有效位为0，代表该虚拟页并没有被缓存在物理内存中。**虚拟页没有被缓存在物理内存中（缓存未命中）被称为缺页。**

当CPU遇见缺页时会触发一个缺页异常，缺页异常将控制权转向操作系统内核，然后调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，如果牺牲页已被修改过，内核会先将它复制回硬盘（采用写回机制而不是直写也是为了尽量减少对硬盘的访问次数），然后再把该虚拟页覆盖到牺牲页的位置，并且更新PTE。

当缺页异常处理程序返回时，它会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重新发送给MMU。由于现在已经成功处理了缺页异常，所以最终结果是页命中，并得到物理地址。

这种在硬盘和内存之间传送页的行为称为页面调度（paging）：页从硬盘换入内存和从内存换出到硬盘。当缺页异常发生时，才将页面换入到内存的策略称为按需页面调度（demand paging），所有现代操作系统基本都使用的是按需页面调度的策略。

虚拟内存跟CPU高速缓存（或其他使用缓存的技术）一样依赖于局部性原则。虽然处理缺页消耗的性能很多（毕竟还是要从硬盘中读取），而且程序在运行过程中引用的不同虚拟页的总数可能会超出物理内存的大小，但是**局部性原则保证了在任意时刻，程序将趋向于在一个较小的活动页面（active page）集合上工作，这个集合被称为工作集（working set）**。根据空间局部性原则（一个被访问过的内存地址以及其周边的内存地址都会有很大几率被再次访问）与时间局部性原则（一个被访问过的内存地址在之后会有很大几率被再次访问），只要将工作集缓存在物理内存中，接下来的地址翻译请求很大几率都在其中，从而减少了额外的硬盘流量。

如果一个程序没有良好的局部性，将会使工作集的大小不断膨胀，直至超过物理内存的大小，这时程序会产生一种叫做抖动（thrashing）的状态，页面会不断地换入换出，如此多次的读写硬盘开销，性能自然会十分“恐怖”。**所以，想要编写出性能高效的程序，首先要保证程序的时间局部性与空间局部性。**

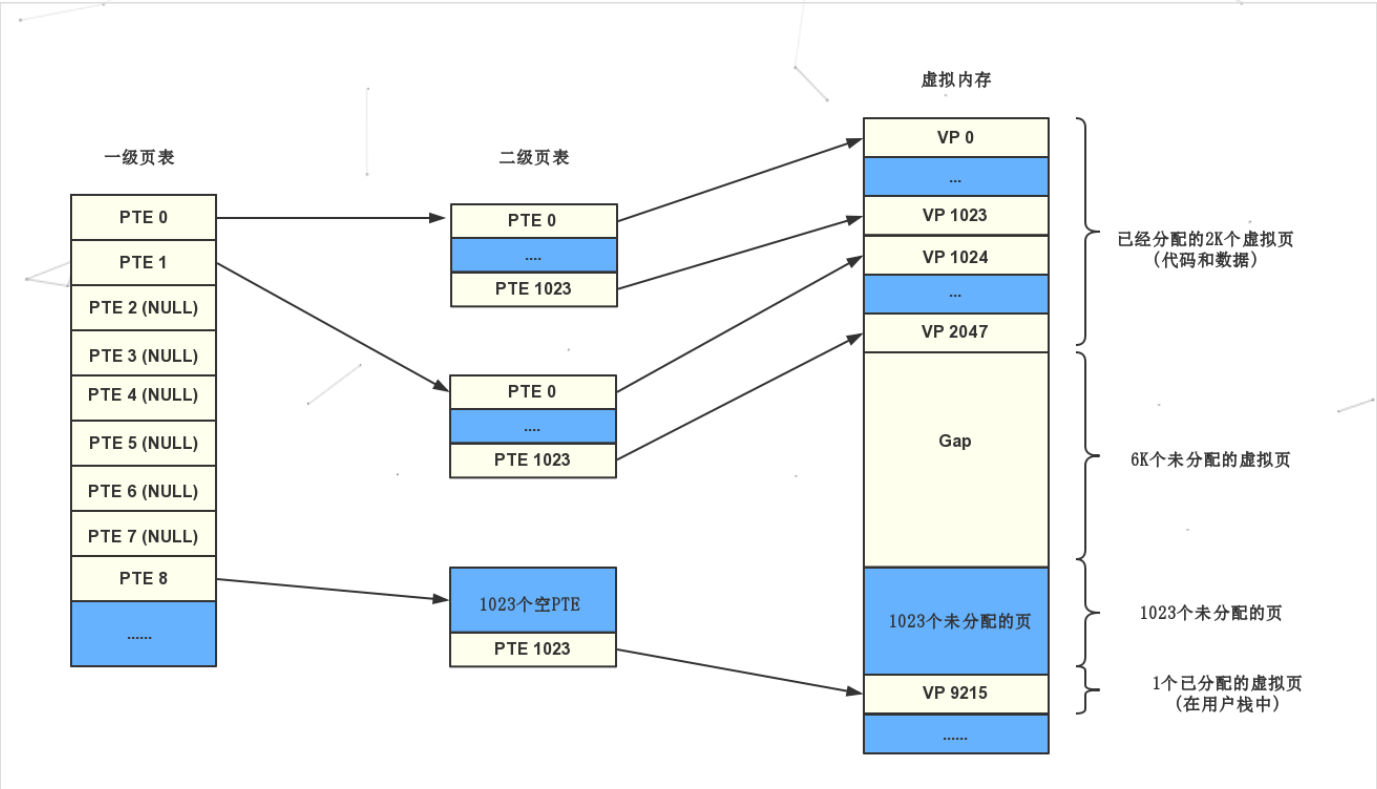
多级页表

我们目前为止讨论的只是单页表，但在实际的环境中虚拟空间地址都是很大的（一个32位系统的地址空间有 2^32 = 4GB，更别说64位系统了）。在这种情况下，使用一个单页表明显是效率低下的。

常用方法是使用层次结构的页表。假设我们的环境为一个32位的虚拟地址空间，它有如下形式：

- 虚拟地址空间被分为4KB的页，每个PTE都是4字节。
- 内存的前2K个页面分配给了代码和数据。
- 之后的6K个页面还未被分配。
- 再接下来的1023个页面也未分配，其后的1个页面分配给了用户栈。

下图是为该虚拟地址空间构造的二级页表层次结构（真实情况中多为四级或更多），一级页表（1024个PTE正好覆盖4GB的虚拟地址空间，同时每个PTE只有4字节，这样一级页表与二级页表的大小也正好与一个页面的大小一致都为4KB）的每个PTE负责映射虚拟地址空间中一个4MB的片（chunk），每一片都由1024个连续的页面组成。二级页表中的每个PTE负责映射一个4KB的虚拟内存页面。



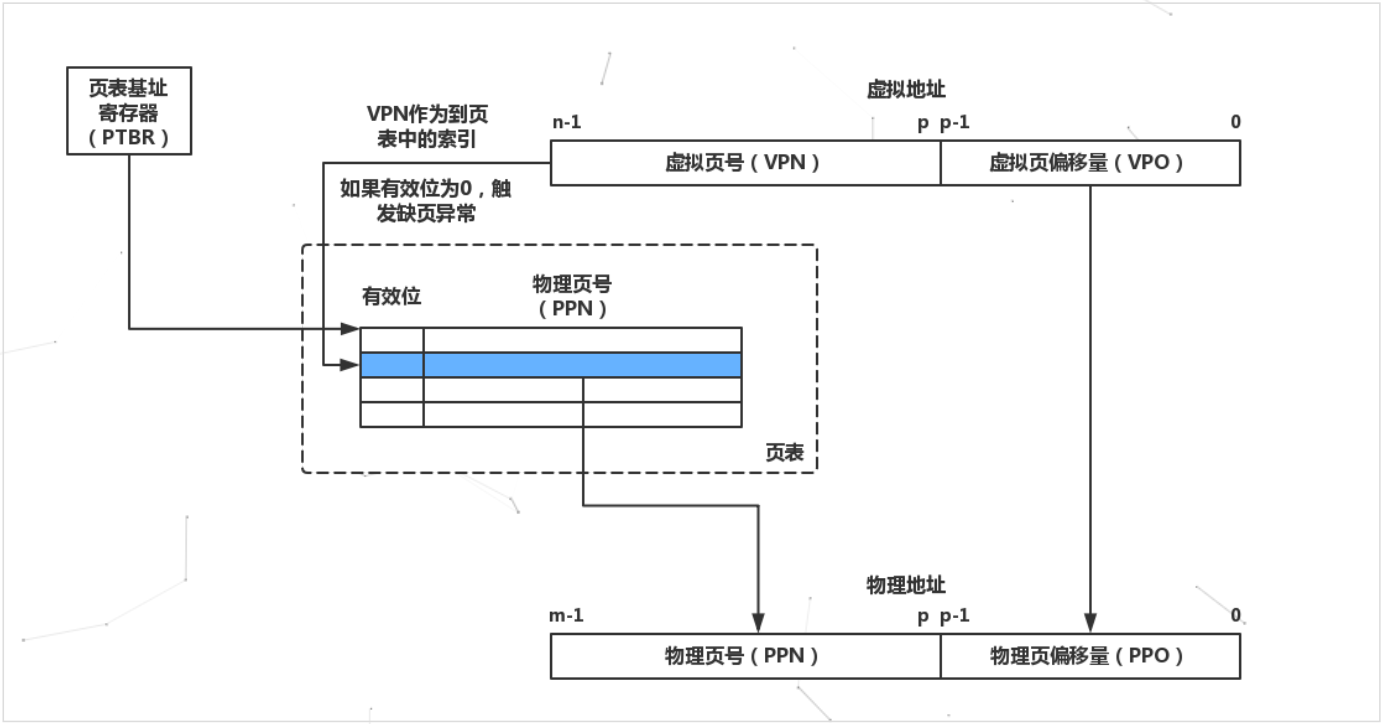
这个结构看起来很像是一个 B-Tree，这种层次结构有效的减缓了内存要求：

- 如果一个一级页表的一个PTE是空的，那么相应的二级页表也不会存在。这代表一种巨大的潜在节约（对于一个普通的程序来说，虚拟地址空间的大部分都会是未分配的）。
- 只有一级页表才总是需要缓存在内存中的，这样虚拟内存系统就可以在需要时创建、页面调入或调出二级页表（只有经常使用的二级页表才会被缓存在内存中），这就减少了内存的压力。

地址翻译的过程

从形式上来说，**地址翻译是一个N元素的虚拟地址空间中的元素和一个M元素的物理地址空间中元素之间的映射。**

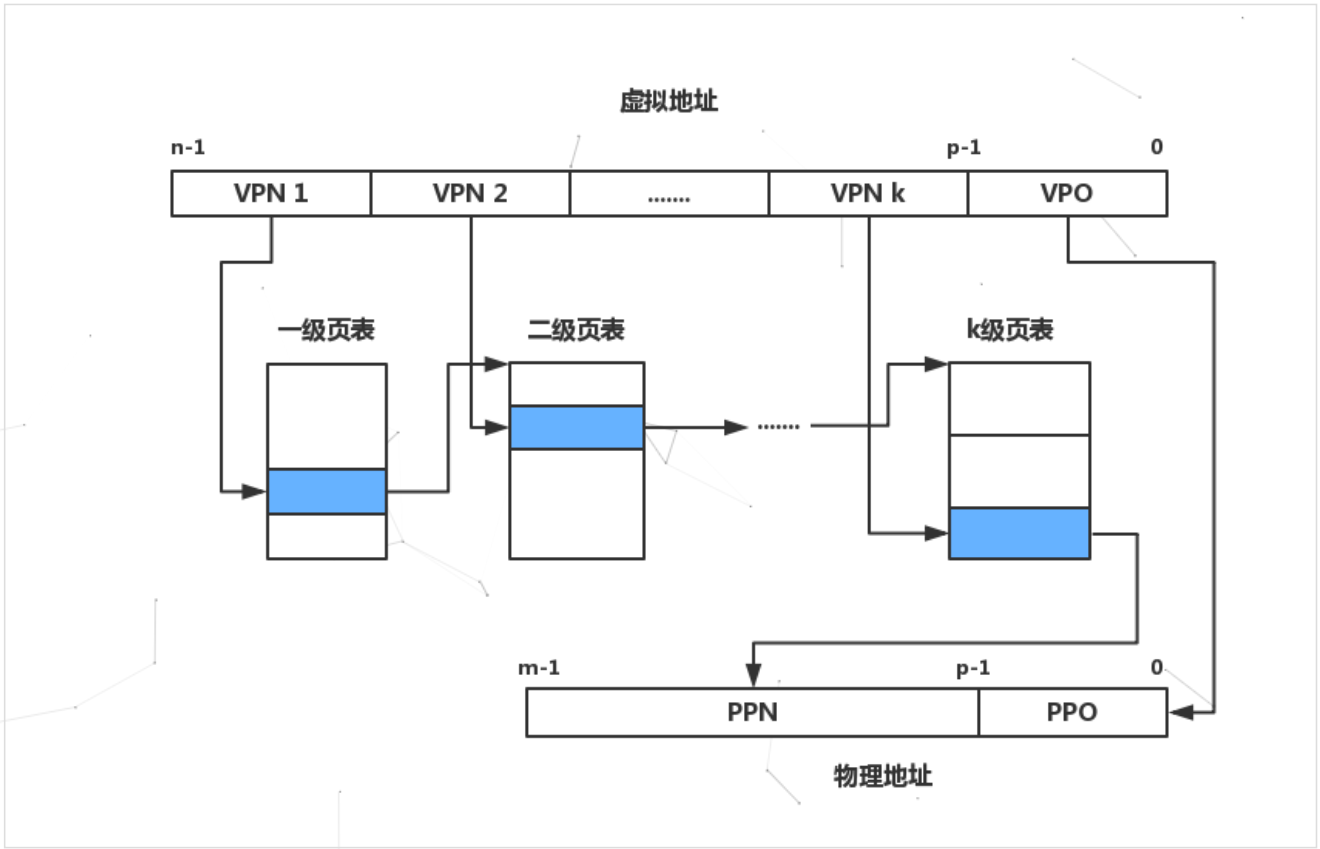
下图为MMU利用页表进行寻址的过程：



页表基址寄存器（PTBR）指向当前页表。**一个n位的虚拟地址包含两个部分，一个p位的虚拟页面偏移量（Virtual Page Offset, VPO）和一个（n - p）位的虚拟页号（Virtual Page Number, VPN）。**

MMU根据VPN来选择对应的PTE，例如 `VPN 0` 代表 `PTE 0` 、 `VPN 1` 代表 `PTE 1` ....因为物理页与虚拟页的大小是一致的，所以物理页面偏移量（Physical Page Offset, PPO）与VPO是相同的。那么之后**只要将PTE中的物理页号（Physical Page Number, PPN）与虚拟地址中的VPO串联起来，就能得到相应的物理地址。**

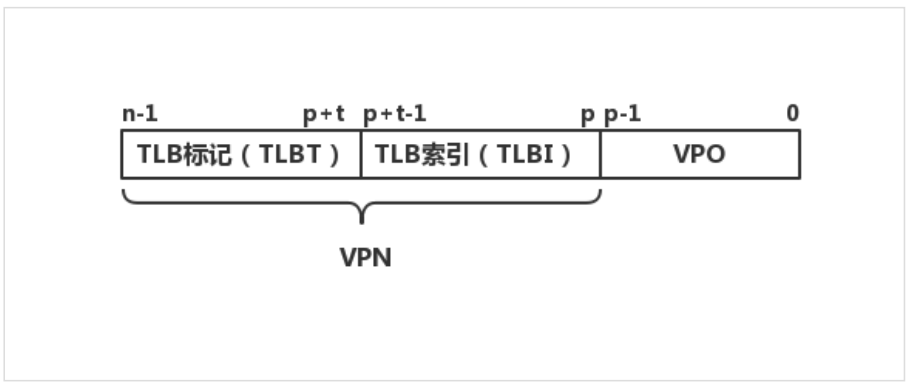
多级页表的地址翻译也是如此，只不过因为有多个层次，所以VPN需要分成多段。**假设有一个k级页表，虚拟地址会被分割成k个VPN和1个VPO，每个 `VPN i` 都是一个到第i级页表的索引。**为了构造物理地址，MMU需要访问k个PTE才能拿到对应的PPN。



TLB

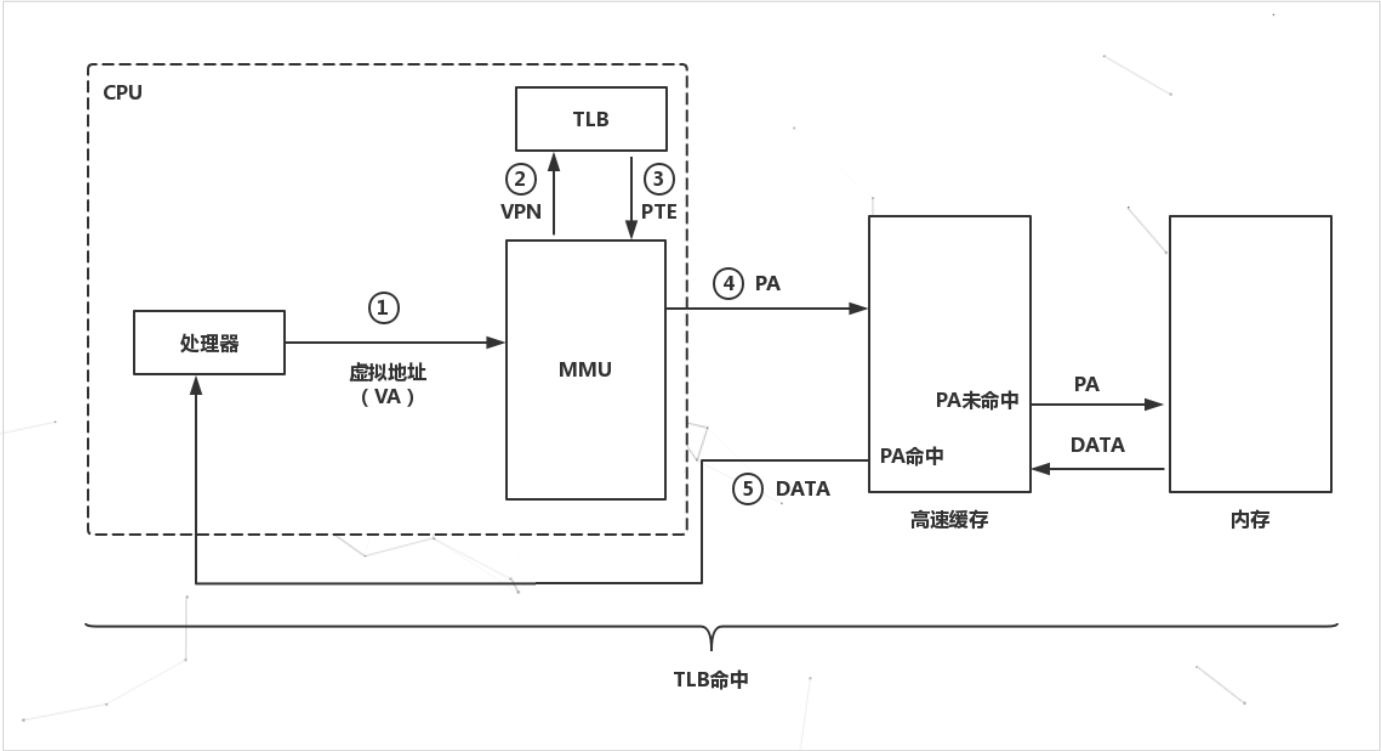
页表是被缓存在内存中的，尽管内存的速度相对于硬盘来说已经非常快了，但与CPU还是有所差距。**为了防止每次地址翻译操作都需要去访问内存，CPU使用了高速缓存与TLB来缓存PTE。**

在最糟糕的情况下（不包括缺页），MMU需要访问内存取得相应的PTE，这个代价大约为几十到几百个周期，如果PTE凑巧缓存在L1高速缓存中（如果L1没有还会从L2中查找，不过我们忽略多级缓冲区的细节），那么性能开销就会下降到1个或2个周期。然而，许多系统甚至需要消除即使这样微小的开销，TLB由此而生。



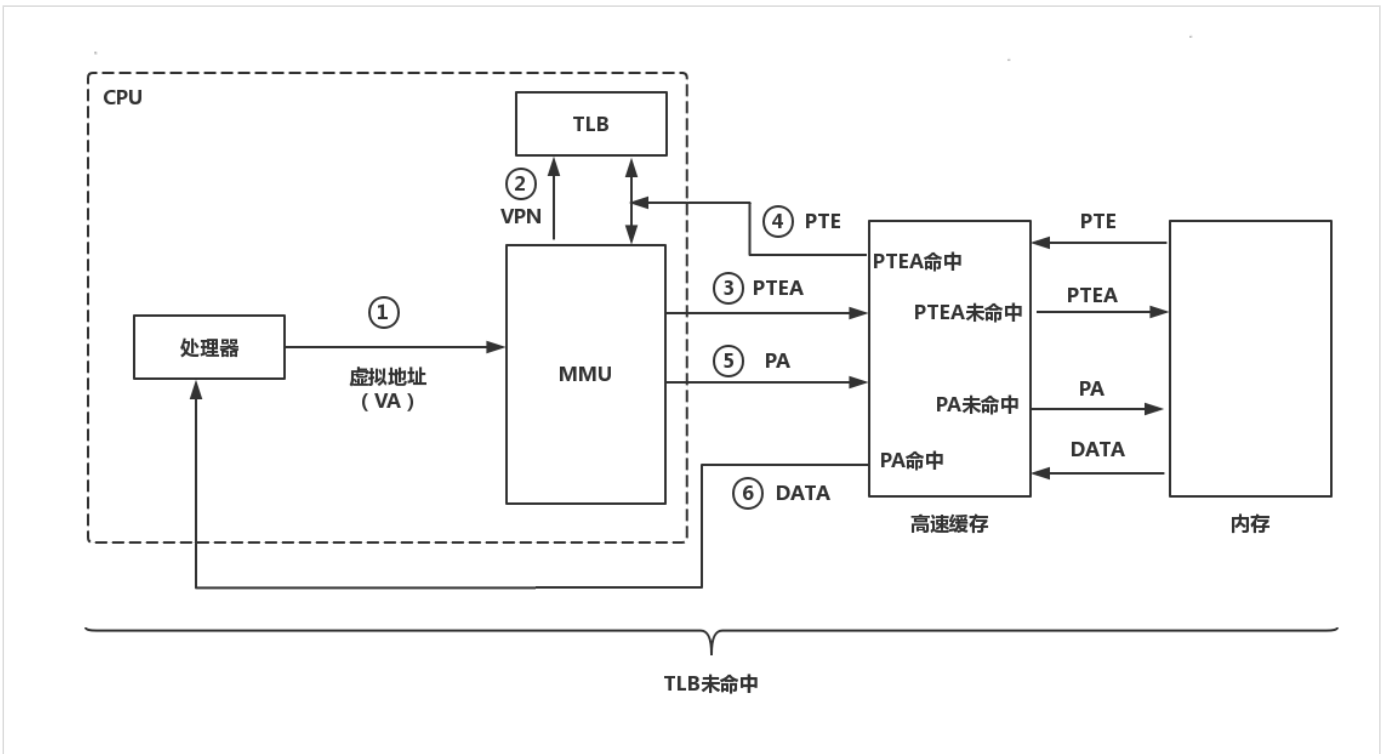
TLB（Translation Lookaside Buffer, TLB）被称为翻译后备缓冲器或翻译旁路缓冲器，它是**MMU中的一个缓冲区，其中每一行都保存着一个由单个PTE组成的块。用于组选择和行匹配的索引与标记字段是从VPN中提取出来的，如果TLB中有 T = 2^t 个组，那么TLB索引（TLBI）是由VPN的t个最低位组成的，而TLB标记（TLBT）是由VPN中剩余的位组成的。**

下图为地址翻译的流程（TLB命中的情况下）：



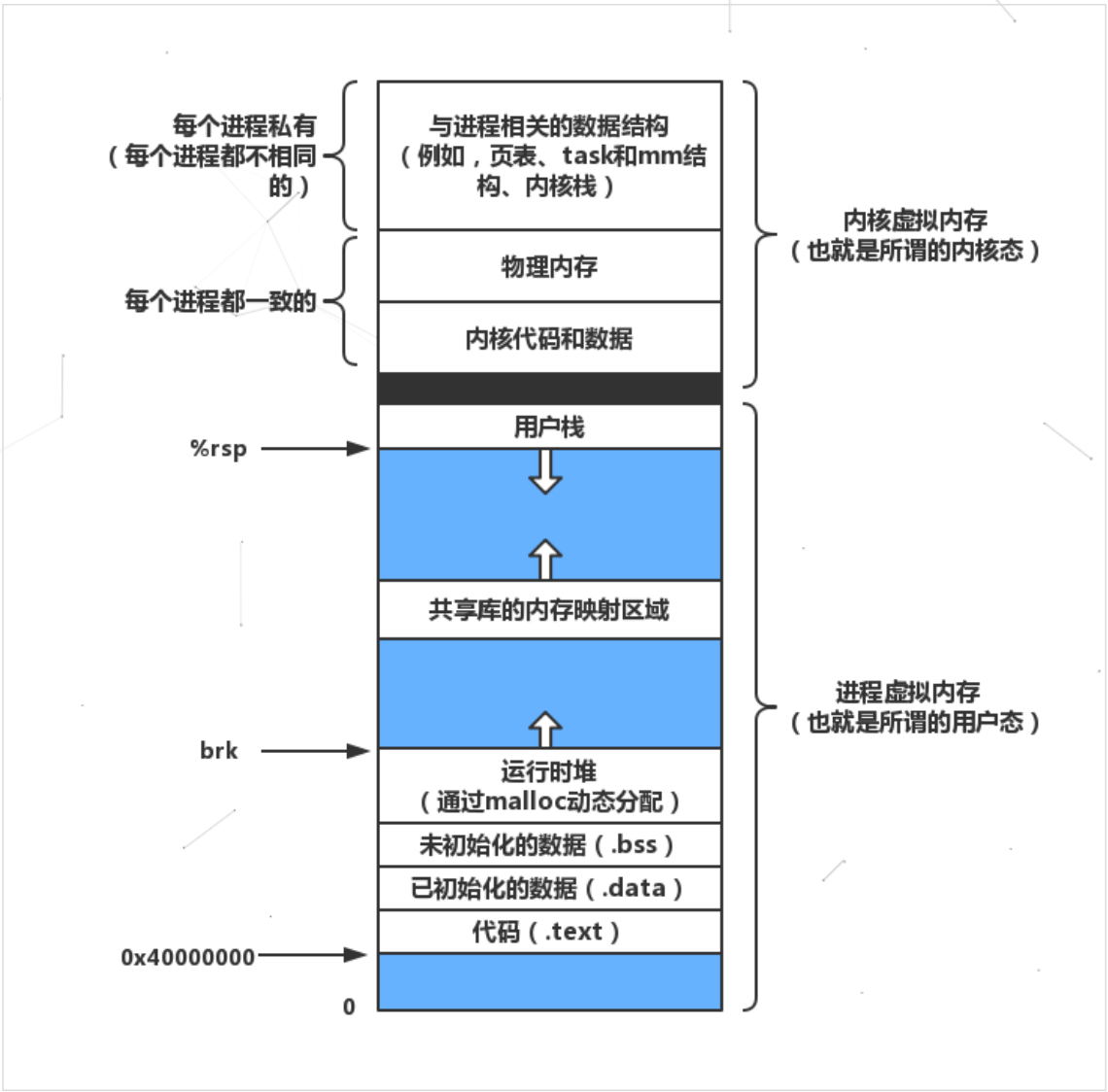
- 第一步，CPU将一个虚拟地址交给MMU进行地址翻译。
- 第二步和第三步，MMU通过TLB取得相应的PTE。
- 第四步，MMU通过PTE翻译出物理地址并将它发送给高速缓存/内存。
- 第五步，高速缓存返回数据到CPU（如果缓存命中的话，否则还需要访问内存）。

当TLB未命中时，MMU必须从高速缓存/内存中取出相应的PTE，并将新取得的PTE存放到TLB（如果TLB已满会覆盖一个已经存在的PTE）。



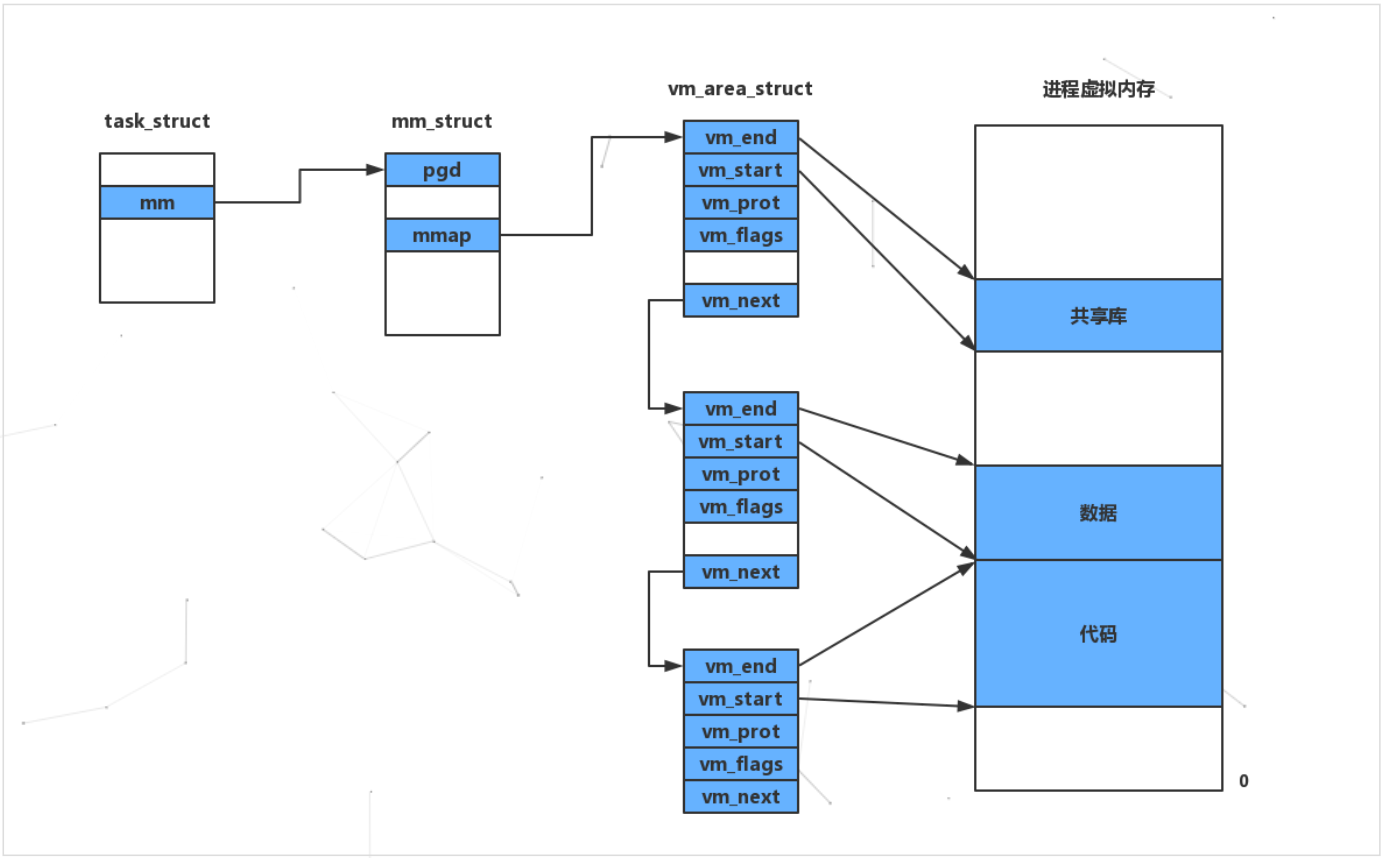
Linux中的虚拟内存系统

Linux为每个进程维护了一个单独的虚拟地址空间。虚拟地址空间分为内核空间与用户空间，用户空间包括代码、数据、堆、共享库以及栈，内核空间包括内核中的代码和数据结构，内核空间的某些区域被映射到所有进程共享的物理页面。Linux也将一组连续的虚拟页面（大小等于内存总量）映射到相应的一组连续的物理页面，这种做法为内核提供了一种便利的方法来访问物理内存中任何特定的位置。



Linux将虚拟内存组织成一些区域（也称为段）的集合，区域的概念允许虚拟地址空间有间隙。一个区域就是已经存在着的已分配的虚拟内存的连续片（chunk）。例如，代码段、数据段、堆、共享库段，以及用户栈都属于不同的区域，每个存在的虚拟页都保存在某个区域中，而不属于任何区域的虚拟页是不存在的，也不能被进程所引用。

内核为系统中的每个进程维护一个单独的任务结构（task_struct）。任务结构中的元素包含或者指向内核运行该进程所需的所有信息（PID、指向用户栈的指针、可执行目标文件的名字、程序计数器等）。



- mm_struct：描述了虚拟内存的当前状态。pgd指向一级页表的基址（当内核运行这个进程时，pgd会被存放在CR3控制寄存器，也就是页表基址寄存器中），mmap指向一个vm_area_structs的链表，其中每个vm_area_structs都描述了当前虚拟地址空间的一个区域。
- vm_starts：指向这个区域的起始处。
- vm_end：指向这个区域的结束处。
- vm_prot：描述这个区域内包含的所有页的读写许可权限。
- vm_flags：描述这个区域内的页面是与其他进程共享的，还是这个进程私有的以及一些其他信息。
- vm_next：指向链表的下一个区域结构。

内存映射

Linux通过将一个虚拟内存区域与一个硬盘上的文件关联起来，以初始化这个虚拟内存区域的内容，这个过程称为内存映射（memory mapping）。这种将虚拟内存系统集成到文件系统的方法可以简单而高效地把程序和数据加载到内存中。

一个区域可以映射到一个普通硬盘文件的连续部分，例如一个可执行目标文件。文件区（section）被分成页大小的片，每一片包含一个虚拟页的初始内容。**由于按需页面调度的策略，这些虚拟页面没有实际交换进入物理内存，直到CPU引用的虚拟地址在该区域的范围内**。如果区域比文件区要大，那么就用零来填充这个区域的余下部分。

一个区域也可以映射到一个匿名文件，匿名文件是由内核创建的，包含的全是二进制零。当CPU第一次引用这样一个区域内的虚拟页面时，内核就在物理内存中找到一个合适的牺牲页面，如果该页面被修改过，就先将它写回到硬盘，之后用二进制零覆盖牺牲页并更新页表，将这个页面标记为已缓存在内存中的。

简单的来说：**普通文件映射就是将一个文件与一块内存建立起映射关系，对该文件进行IO操作可以绕过内核直接在用户态完成（用户态在该虚拟地址区域读写就相当于读写这个文件）。匿名文件映射一般在用户空间需要分配一段内存来存放数据时，由内核创建匿名文件并与内存进行映射，之后用户态就可以通过操作这段虚拟地址来操作内存了。匿名文件映射最熟悉的应用场景就是动态内存分配（malloc()函数）。**

Linux很多地方都采用了“懒加载”机制，自然也包括内存映射。不管是普通文件映射还是匿名映射，Linux只会先划分虚拟内存地址。只有当CPU第一次访问该区域内的虚拟地址时，才会真正的与物理内存建立映射关系。

只要虚拟页被初始化了，它就在一个由内核维护的交换文件（swap file）之间换来换去。交换文件又称为交换空间（swap space）或交换区域（swap area）。swap区域不止用于页交换，在物理内存不够的情况下，还会将部分内存数据交换到swap区域（使用硬盘来扩展内存）。

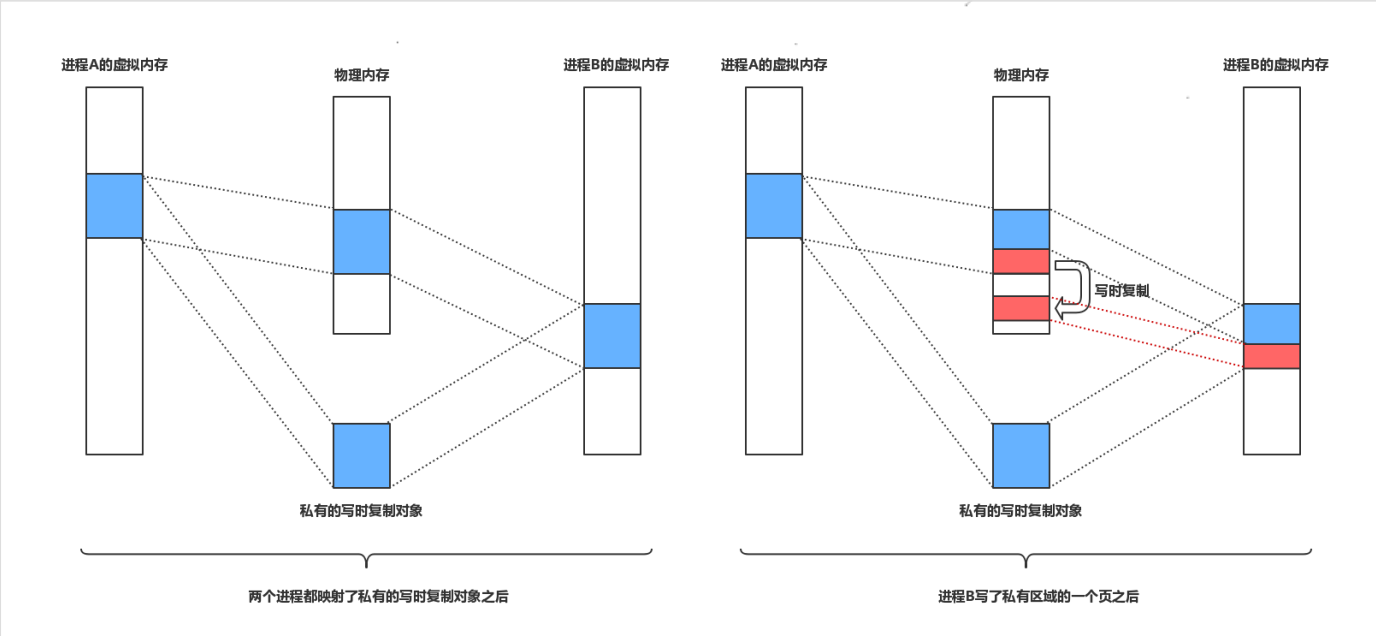
共享对象

虚拟内存系统为每个进程提供了私有的虚拟地址空间，这样可以保证进程之间不会发生错误的读写。但多个进程之间也含有相同的部分，例如每个C程序都使用到了C标准库，如果每个进程都在物理内存中保持这些代码的副本，那会造成很大的内存资源浪费。

内存映射提供了共享对象的机制，来避免内存资源的浪费。一个对象被映射到虚拟内存的一个区域，要么是作为共享对象，要么是作为私有对象的。

如果一个进程将一个共享对象映射到它的虚拟地址空间的一个区域内，那么这个进程对这个区域的任何写操作，对于那些也把这个共享对象映射到它们虚拟内存的其他进程而言，也是可见的。相对的，对一个映射到私有对象的区域的任何写操作，对于其他进程来说是不可见的。一个映射到共享对象的虚拟内存区域叫做共享区域，类似地，也有私有区域。

为了节约内存，私有对象开始的生命周期与共享对象基本上是一致的（在物理内存中只保存私有对象的一份副本），并使用写时复制的技术来应对多个进程的写冲突。



只要没有进程试图写它自己的私有区域，那么多个进程就可以继续共享物理内存中私有对象的一个单独副本。然而，只要有一个进程试图对私有区域的某一页面进行写操作，就会触发一个保护异常。在上图中，进程B试图对私有区域的一个页面进行写操作，该操作触发了保护异常。**异常处理程序会在物理内存中创建这个页面的一个新副本，并更新PTE指向这个新的副本，然后恢复这个页的可写权限。**

还有一个典型的例子就是 `fork()` 函数，该函数用于创建子进程。当 `fork()` 函数被当前进程调用时，内核会为新进程创建各种必要的数据结构，并分配给它一个唯一的PID。为了给新进程创建虚拟内存，它复制了当前进程的 `mm_struct` 、 `vm_area_struct` 和页表的原样副本。并将两个进

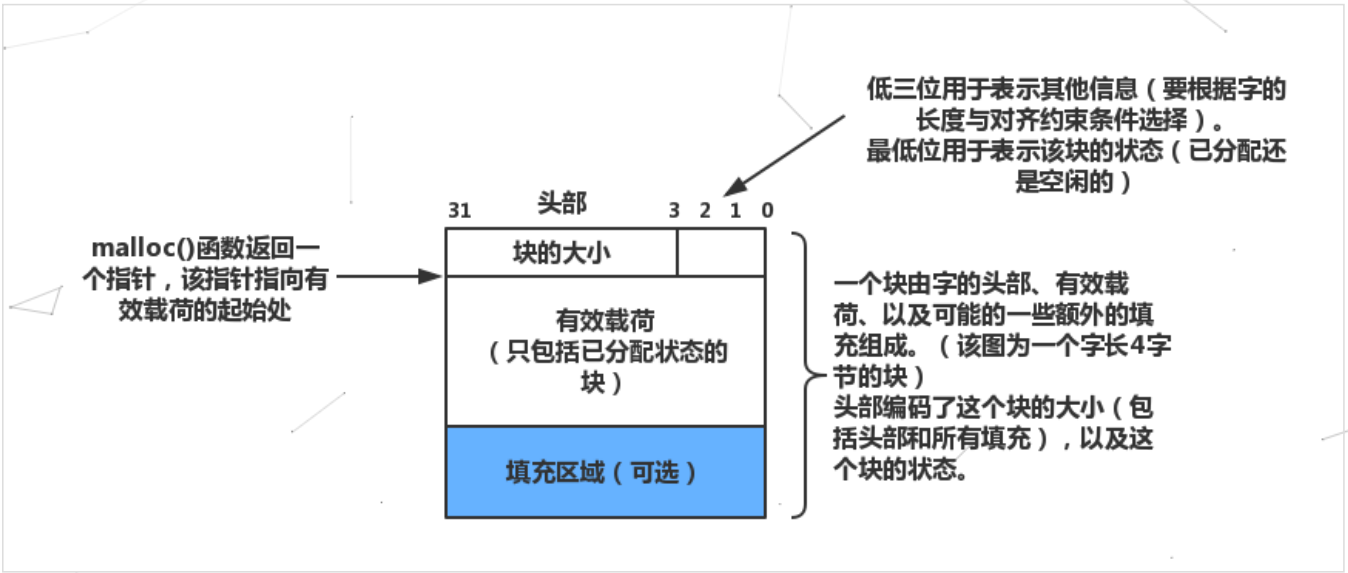
程的每个页面都标为只读，两个进程中的每个区域都标记为私有区域（写时复制）。

这样，父进程和子进程的虚拟内存空间完全一致，只有当这两个进程中的任一个进行写操作时，再使用写时复制来保证每个进程的虚拟地址空间私有的抽象概念。

动态内存分配

虽然可以使用内存映射（`mmap()`函数）来创建和删除虚拟内存区域来满足运行时动态内存分配的问题。然而，为了更好的移植性与便利性，还需要一个更高层面的抽象，也就是动态内存分配器（dynamic memory allocator）。

动态内存分配器维护着一个进程的虚拟内存区域，也就是我们所熟悉的“堆（heap）”，内核中还维护着一个指向堆顶的指针brk（break）。动态内存分配器将堆视为一个连续的虚拟内存块（chunk）的集合，每个块有两种状态，已分配和空闲。已分配的块显式地保留为供应用程序使用，空闲块则可以用来进行分配，它的空闲状态直到它显式地被应用程序分配为止。已分配的块要么被应用程序显式释放，要么被垃圾回收器所释放。



本文只讲解动态内存分配的一些概念，关于动态内存分配器的实现已经超出了本文的讨论范围。如果有对它感兴趣的同学，可以去参考dlmalloc的源码，它是由Doug Lea（就是写Java并发包的那位）实现的一个设计巧妙的内存分配器，而且源码中的注释十分多。

内存碎片

造成堆的空间利用率很低的主要原因是一种被称为碎片（fragmentation）的现象，当虽然有未使用的内存但这块内存并不能满足分配请求时，就会产生碎片。有以下两种形式的碎片：

- 内部碎片：在一个已分配块比有效载荷大时发生。例如，程序请求一个5字（这里我们不纠结字的大小，假设一个字为4字节，堆的大小为16字并且要保证边界双字对齐）的块，内存分配器为了保证空闲块是双字边界对齐的（具体实现中对齐的规定可能略有不同，但对齐是肯定会有的），只好分配一个6字的块。在本例中，已分配块为6字，有效载荷为5字，内部碎片为已分配块减去有效载荷，为1字。

- 外部碎片：当空闲内存合计起来足够满足一个分配请求，但是没有一个单独的空闲块足够大到可以来处理这个请求时发生。外部碎片难以量化且不可预测，所以分配器通常采用启发式策略来试图维持少量的大空闲块，而不是维持大量的小空闲块。分配器也会根据策略与分配请求的匹配来分割空闲块与合并空闲块（必须相邻）。

空闲链表

分配器将堆组织为一个连续的已分配块和空闲块的序列，该序列被称为空闲链表。空闲链表分为隐式空闲链表与显式空闲链表。

- 隐式空闲链表，是一个单向链表，并且每个空闲块仅仅是通过头部中的大小字段隐含地连接着的。
- 显式空闲链表，即是将空闲块组织为某种形式的显式数据结构（为了更加高效地合并与分割空闲块）。例如，将堆组织为一个双向空闲链表，在每个空闲块中，都包含一个前驱节点的指针与后继节点的指针。

查找一个空闲块一般有如下几种策略：

- 首次适配：从头开始搜索空闲链表，选择第一个遇见的合适的空闲块。它的优点在于趋向于将大的空闲块保留在链表的后面，缺点是它趋向于在靠近链表前部处留下碎片。
- 下一次适配：每次从上一次查询结束的地方开始进行搜索，直到遇见合适的空闲块。这种策略通常比首次适配效率高，但是内存利用率则要低得多了。
- 最佳适配：检查每个空闲块，选择适合所需请求大小的最小空闲块。最佳适配的内存利用率是三种策略中最高的，但它需要对堆进行彻底的搜索。

对一个链表进行查找操作的效率是线性的，为了减少分配请求对空闲块匹配的时间，**分配器通常采用分离存储（segregated storage）的策略，即是维护多个空闲链表，其中每个链表的块有大致相等的大小。**

一种简单的分离存储策略：分配器维护一个空闲链表数组，然后将所有可能的块分成一些等价类（也叫做大小类（size class）），每个大小类代表一个空闲链表，并且每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小（例如，某个大小类的范围定义为（17~32），那么这个空闲链表全由大小为32的块组成）。

当有一个分配请求时，我们检查相应的空闲链表。如果链表非空，那么就分配其中第一块的全部。如果链表为空，分配器就向操作系统请求一个固定大小的额外内存片，将这个片分成大小相等的块，然后将这些块链接起来形成新的空闲链表。

要释放一个块，分配器只需要简单地将这个块插入到相应的空闲链表的头部。

垃圾回收

在编写C程序时，一般只能显式地分配与释放堆中的内存（`malloc()` 与 `free()`），程序员不仅需要分配内存，还需要负责内存的释放。

许多现代编程语言都内置了自动内存管理机制（通过引入自动内存管理库也可以让C/C++实现自动内存管理），**所谓自动内存管理，就是自动判断不再需要的堆内存（被称为垃圾内存），然后自动释放这些垃圾内存。**

自动内存管理的实现是垃圾收集器（garbage collector），它是一种动态内存分配器，它会自动释放应用程序不再需要的已分配块。

垃圾收集器一般采用以下两种（之一）的策略来判断一块堆内存是否为垃圾内存：

- 引用计数器：在数据的物理空间中添加一个计数器，当有其他数据与其相关时（引用），该计数器加一，反之则减一。通过定期检查计数器的值，只要为0则认为是垃圾内存，可以释放它所占用的已分配块。使用引用计数器，实现简单直接，但缺点也很明显，它无法回收循环引用的两个对象（假设有对象A与对象B，它们2个互相引用，但实际上对象A与对象B都已经是没用的对象了）。

- 可达性分析：垃圾收集器将堆内存视为一张有向图，然后选出一组根节点（例如，在Java中一般为类加载器、全局变量、运行时常量池中的引用类型变量等），根节点必须是足够“活跃“的对象。然后计算从根节点集合出发的可达路径，只要从根节点出发不可达的节点，都视为垃圾内存。

垃圾收集器进行回收的算法有如下几种：

- 标记-清除：该算法分为标记（mark）和清除（sweep）两个阶段。首先标记出所有需要回收的对象，然后在标记完成后统一回收所有被标记的对象。标记-清除算法实现简单，但它的效率不高，而且会产生许多内存碎片。

- 标记-整理：标记-整理与标记-清除算法基本一致，只不过后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉边界以外的内存。

- 复制：**将程序所拥有的内存空间划分为大小相等的两块，每次都只使用其中的一块。当这一块的内存用完了，就把还存活着的对象复制到另一块内存上，然后将已使用过的内存空间进行清理**。这种方法不必考虑内存碎片问题，但内存利用率很低。这个比例不是绝对的，像HotSpot虚拟机为了避免浪费，将内存划分为Eden空间与两个Survivor空间，每次都只使用Eden和其中一个Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一个Survivor空间上，然后清理掉Eden和刚才使用过的Survivor空间。HotSpot虚拟机默认的Eden和Survivor的大小比例为8：1，只有10%的内存空间会被闲置浪费。

- 分代：**分代算法根据对象的存活周期的不同将内存划分为多块，这样就可以对不同的年代采用不同的回收算法**。一般分为新生代与老年代，新生代存放的是存活率较低的对象，可以采用复制算法；老年代存放的是存活率较高的对象，如果使用复制算法，那么内存空间会不够用，所以必须使用标记-清除或标记-整理算法。

总结

虚拟内存是对内存的一个抽象。支持虚拟内存的CPU需要通过虚拟寻址的方式来引用内存中的数据。CPU加载一个虚拟地址，然后发送给MMU进行地址翻译。地址翻译需要硬件与操作系统之间紧密合作，MMU借助页表来获得物理地址。

- 首先，MMU先将虚拟地址发送给TLB以获得PTE（根据VPN寻址）。

- 如果恰好TLB中缓存了该PTE，那么就返回给MMU，否则MMU需要从高速缓存/内存中获得PTE，然后更新缓存到TLB。

- MMU获得了PTE，就可以从PTE中获得对应的PPN，然后结合VPO构造出物理地址。

- 如果在PTE中发现该虚拟页没有缓存在内存，那么会触发一个缺页异常。缺页异常处理程序会把虚拟页缓存进物理内存，并更新PTE。异常处理程序返回后，CPU会重新加载这个虚拟地址，并进行翻译。

虚拟内存系统简化了内存管理、链接、加载、代码和数据的共享以及访问权限的保护：

- 简化链接，独立的地址空间允许每个进程的内存映像使用相同的基本格式，而不管代码和数据实际存放在物理内存的何处。

- 简化加载，虚拟内存使向内存中加载可执行文件和共享对象文件变得更加容易。

- 简化共享，独立的地址空间为操作系统提供了一个管理用户进程和内核之间共享的一致机制。

- 访问权限保护，每个虚拟地址都要经过查询PTE的过程，在PTE中设定访问权限的标记位从而简化内存的权限保护。

操作系统通过将虚拟内存与文件系统结合的方式，来初始化虚拟内存区域，这个过程称为内存映射。应用程序显式分配内存的区域叫做堆，通过动态内存分配器来直接操作堆内存。