CSAPP - 虚拟内存

前言

我们知道计算机会将数据存储在磁盘或者硬盘上, 同时我们使用一个唯一的地址去标识这个位置, 我们也可以通过这个地址来精确的找到我们想要的位置. 我们自然而然的能像到, 如果 CPU 想要知道某一的位置的内容, 那它就可以拿着这个位置的物理地址去寻找, 这种寻址方式就叫**物理寻址**

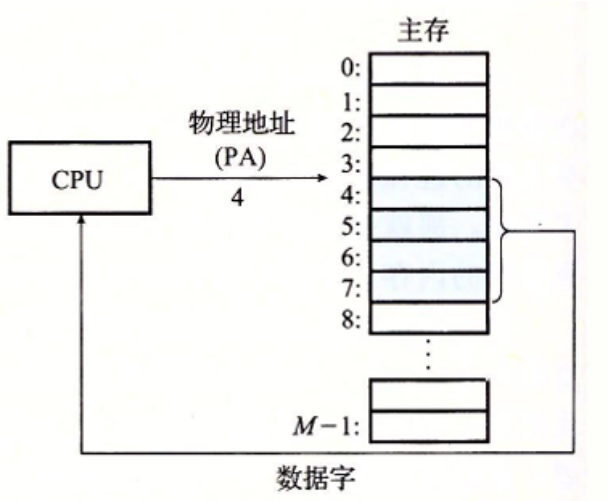

图 9-1　一个使用物理寻址的系统

早期的 PC 会选择物理寻址, 但是现在的处理器会使用一种名叫 **虚拟寻址** 的技术

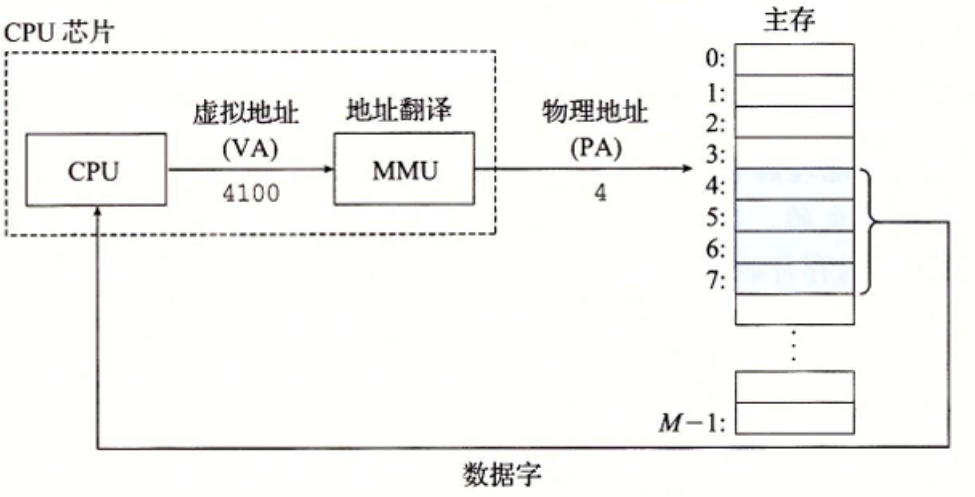

图 9-2　一个使用虚拟寻址的系统

如上图所示,

CPU 不会直接用物理地址去寻找, 而是直接使用虚拟地址, 将虚拟地址交给 MMU 单元进行翻译, MMU 将虚拟地址翻译成物理地址再去缓存中寻找, 找到内容后缓存再讲内容直接返回给 CPU

地址空间

地址空间 (address space) 是一个非负整数地址的有序集合。如果地址空间中整数是连续的，我们说它是 **线性地址空间** (linear address space)。

- 在一个带虚拟内存的系统中，CPU 从一个有 $N = 2^n$ 个地址的地址空间中生成虚拟地址，这个地址空间称为虚拟地址空间 (virtual address space)。一个地址空间大小是由表示最大地址所需要的位数来描述的，如 $N = 2^n$ 个地址的虚拟地址空间叫做 n 位地址空间，现在操作系统支持 32 位或 64 位。
- 一个系统还有 **物理地址空间**，它与系统中物理内存的 $M = 2^m$ (M 不要求是 2 的幂，这里假设为 2 的幂)个字节相对应。

虚拟内存

我们先首先简单说明一下虚拟内存为什么会存在, 以及存在的意义，一个进程在执行的过程中, 进程为认为自己是有一个**连续**的内存空间可以给自己调用的, 该进程的内存的分配和使用都是基于连续来进行的, 但是如果我们电脑上运行的进程多的话, 计算机本身是分配不了那么多连续的内存的, 这就会出现进程无法运行的情况, 但是实际上内存中是有足够的空间来运行这些程序的, 只不过这些地址被分割罢了(也就是不连续), 所以无法直接使用物理地址

这个时候就需要虚拟内存出场了, 他可以将这些物理上零散的位置整合成逻辑上的连续 (实际物理位置是不会变化的) , 这样, 多余的进程就可以使用这些 "连续" 的内存位置了, 至于如何实现的, 大家可以理解为虚拟内存维护了一张映射表，将零散的物理地址映射在了一起,

举个例子 : 物理地址有 $5, 13$, 我维护一张映射表 $1 \Rightarrow 5, 2 \Rightarrow 13$, 那么程序看到的就是 $1, 2$ 这两个连续的数字, 但是实际上使用的是 $5, 13$ 这两个不连续的物理位置

我们知道在实际的存储过程中, 数据是以一页一页的形式存储的, 我们叫 **物理页** , 为了维护虚拟内存中的映射表, 我们也需要有相对应的形式来表示物理页, 于是我们就定义了 **虚拟页** , 虚拟页是一个抽象的概念, 它和物理页的大小完全相同, 有一种专门的数据结构叫**页表条目**, 它存储着它所对应的物理页在虚拟内存的相关情况

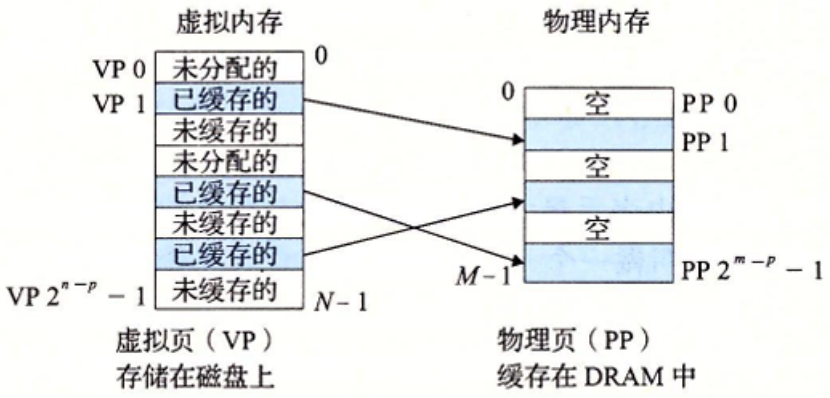

图 9-3　一个 VM 系统是如何使用主存作为缓存的

页表条目大致上存储的信息可以分为三大类

- 当前虚拟页还没有分配实际的物理页
- 当前虚拟页分配了具体的物理页, 但是这个物理页没有缓存
- 当前虚拟页分配了具体的物理页, 但是这个物理页缓存了

虚拟内存作为缓存的工具

页表

页表是一个存放在内存中的**页表条目**(page table entry，`PTE`)的数组，负责将虚拟页映射到物理页

页表条目由**有效位**和 n **位地址字段**组成：

- 有效位表明该虚拟页是否被缓存在`DRAM`中；
- 如果在，地址字段就表示`DRAM`中相应物理页的起始位置；
- 如果不在，地址字段就指向该虚拟页在磁盘上的起始位置；

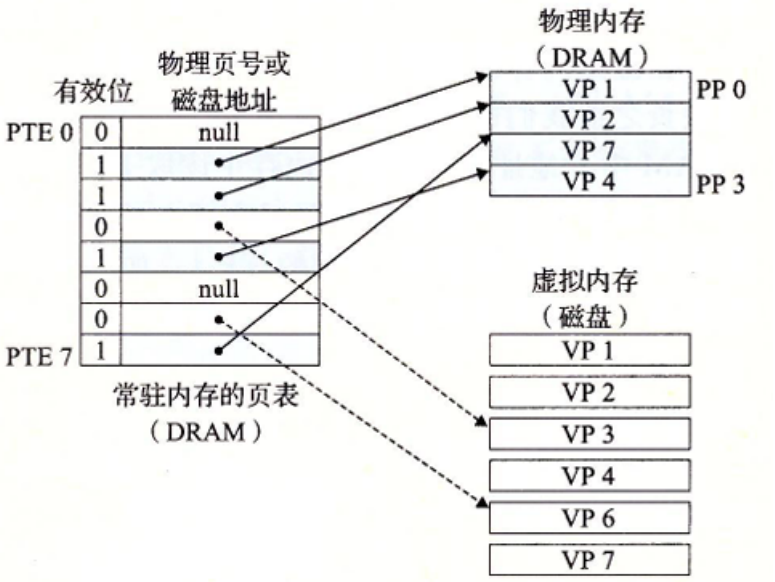

图 9-4　页表

页命中，缺页，分配页面

- **页命中**：一个页命中的过程，就是一个虚拟地址转换为物理地址的过程。

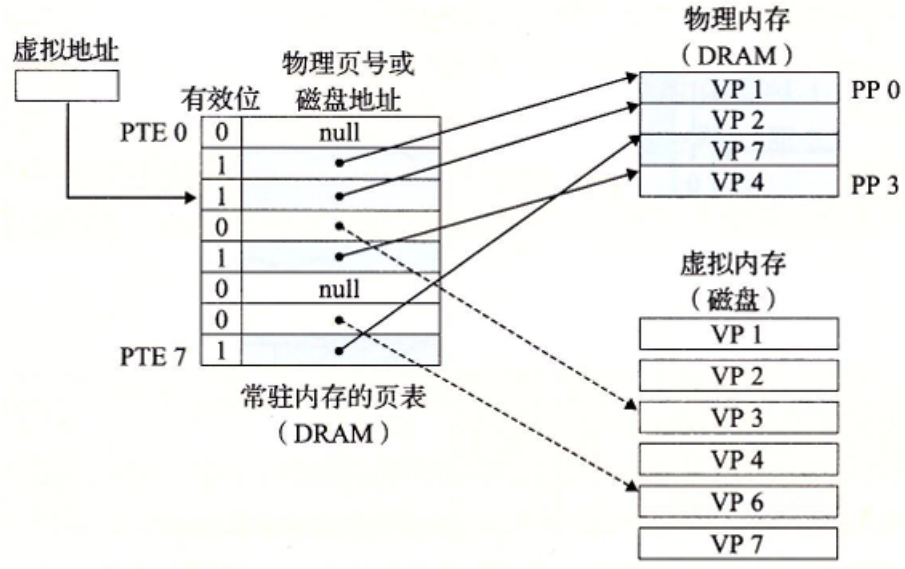

图 9-5　VM 页命中。对 VP 2 中一个字的引用就会命中

- **缺页**：`DRAM`缓存不命中称为缺页 (page fault)，缺页会触发一个缺页异常，异常调用内核中的缺页异常处理程序，在`DRAM`中选择一个牺牲页。

发生缺页的时候, MMU 会发出一个缺页异常, 之后系统会进入内核态, 然后执行缺页异常处理程序. 缺页异常处理程序会操作缓存, 将请求的页读入缓存中, 之后 CPU会再次向 MMU 请求虚拟地址, 这个时候对应的物理页是已经在缓存中的, 就不会再产生缺页异常了, MMU 将地址给缓存, 缓存读出来之后再返回给 CPU, 这个过程结束

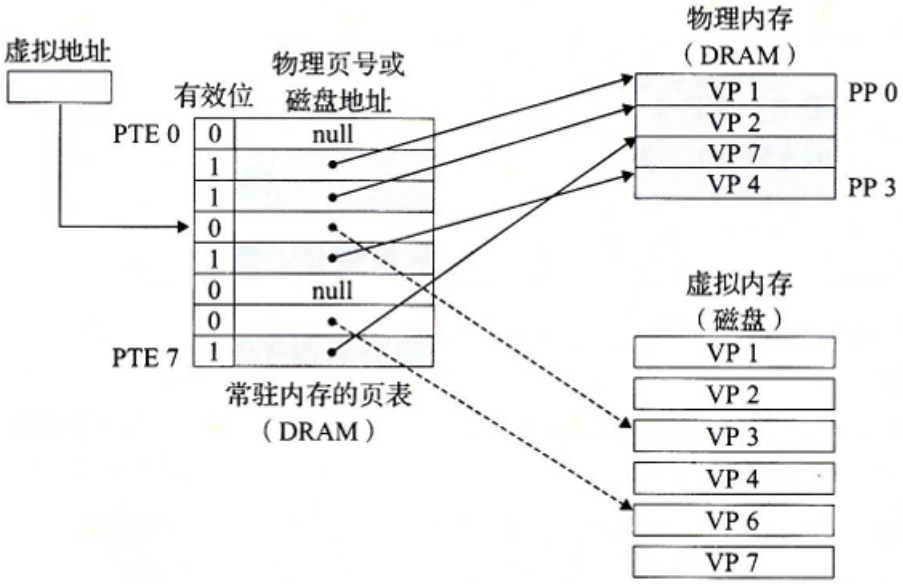

图 9-6　VM 缺页(之前)。对 VP 3 中的字的引用会不命中，从而触发了缺页

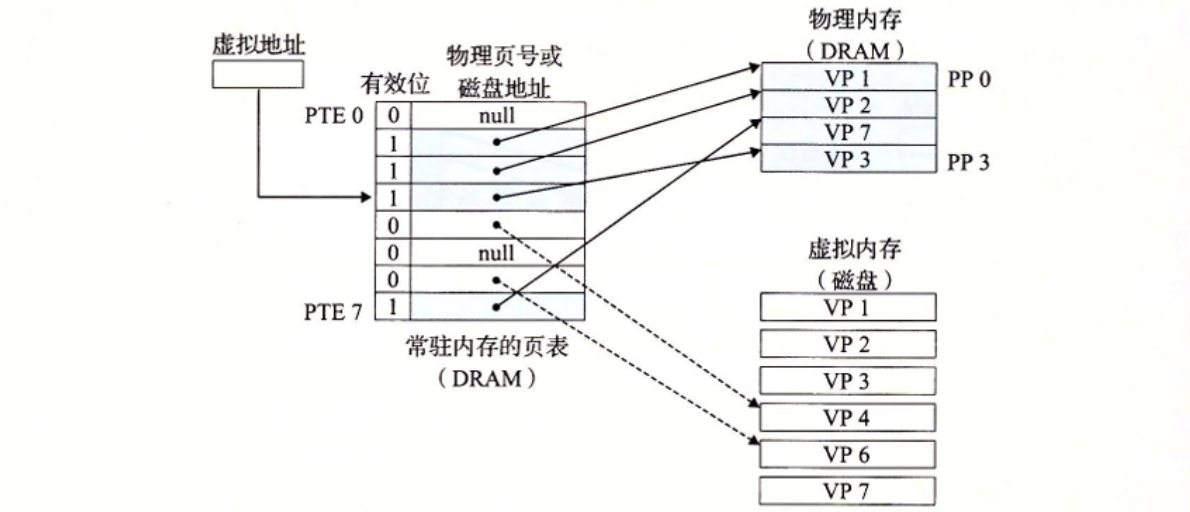

图 9-7　VM 缺页(之后)。缺页处理程序选择 VP 4 作为牺牲页，并从磁盘上用 VP 3 的副本取代它。在缺页处理程序重新启动导致缺页的指令之后，该指令将从内存中正常地读取字，而不会再产生异常

- **分配页面**：操作系统分配一个新的虚拟内存时，如调用`malloc`，首先在磁盘上面创建空间并更新**页表**的`PTE`，使其中某个`PTE`从原来指向`null`，变成指向磁盘上新创建的页面，此时虚拟页从**未分配**状态变为**未缓存**。

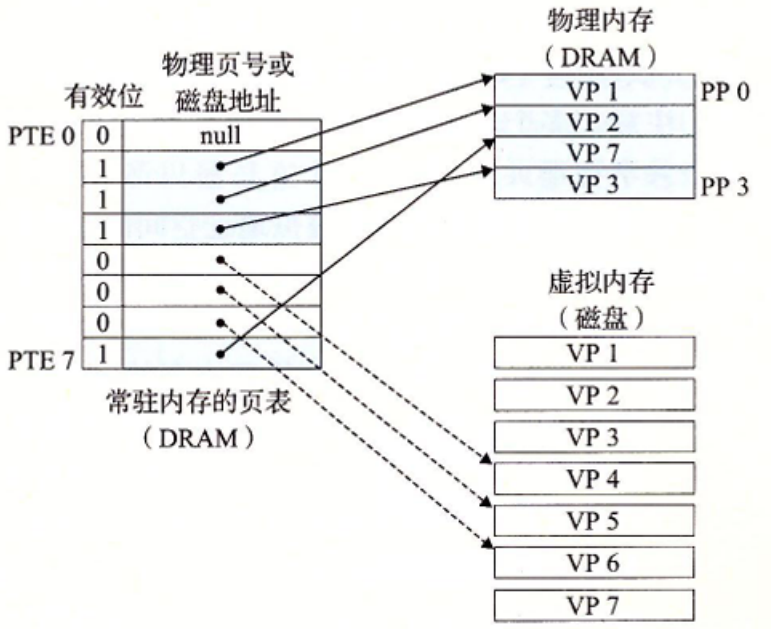

图 9-8　分配一个新的虚拟页面。内核在磁盘上分配 VP 5，并且将 PTE 5 指向这个新的位置

虚拟内存作为内存管理的工具

有一点比较重要的是, 每一个进程都有一个属于自己的页表, 它维护者该进程的映射表, 让他能够正常的工作, 因而每一个进程都有一个独立的虚拟地址空间，且每个进程的虚拟地址空间结构都一样

同时, 虚拟内存还简化了链接和加载, 代码和数据共享的过程 :

- **共享内容** : 如果两个进程使用了共享的内容, 那么他们的页表可以指向同一片区域从而提高效率

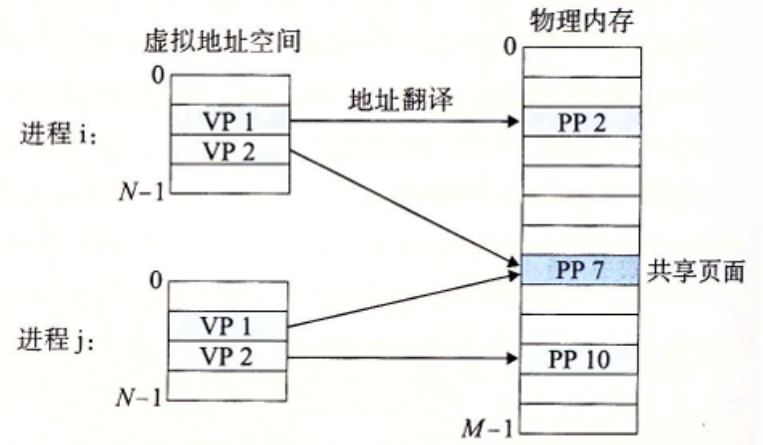

图 9-9　VM 如何为进程提供独立的地址空间。操作系统为系统中的每个进程都维护一个独立的页表

- **简化链接** : 独立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。

  有了虚拟内存的帮助, 每一个进程看自己的内存结构都是一样的 :

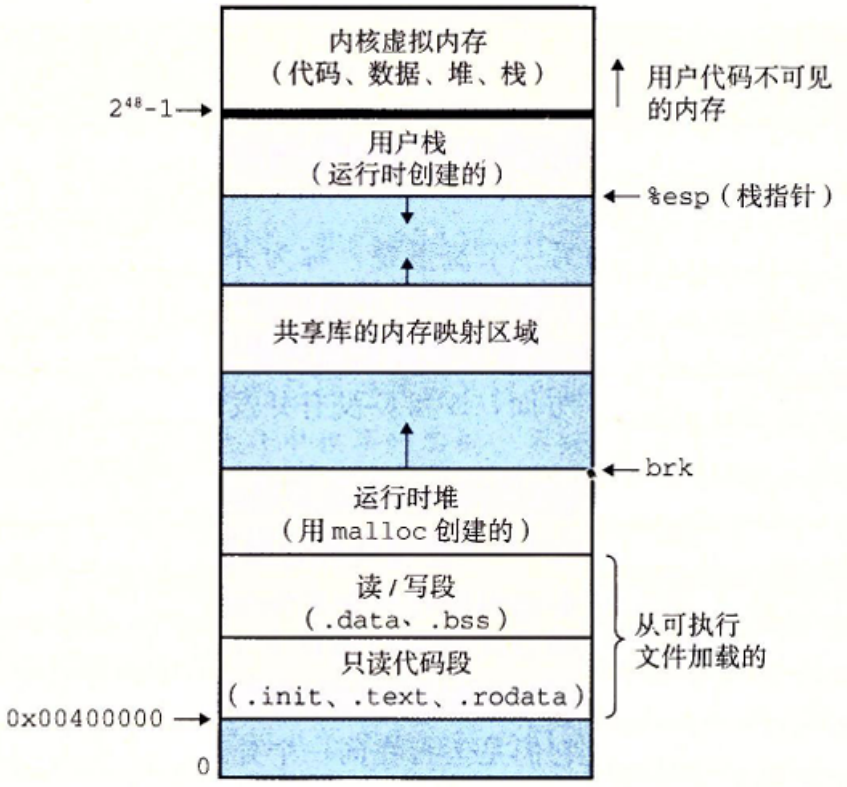

图 8-13　进程地址空间

每一个进程, 代码段总是从 0x400000 开始的, 之后再依次递增

- **简化加载** : 更容易向内存中加载可执行文件和共享对象文件

  加载可执行文件和共享文件，要把目标文件中的 `.text` 和 `.data` 节加载到一个新创建的进程中，Linux加载器分配虚拟页的一个连续的片，从虚拟地址 0x08048000 处开始（32 位），或者从 0x400000 处开始（64 位），加载器为代码段和数据段分配虚拟页，并将其标记为未缓存，将页表条目指向目标文件中适当的位置，加载器不会将任何数据从磁盘复制到内存。

- **简化内存分配** : 当一个运行在用户进程中的程序要求额外的堆内存时(调用 `malloc`)，操作系统会分配 k 个**连续**的虚拟内存页面，并将它们映射到物理内存中**任意**位置的 k 个物理页面。即只需虚拟页面连续，而物理页面随机。

虚拟内存作为内存保护的工具

任何现代计算机系统必须为操作系统提供手段来控制对内存系统的访问。不应该允许个用户进程修改它的只读代码段。而且也不应该允许它读或修改任何内核中的代码和数据结构。不应该允许它读或者写其他进程的私有内存，并且不允许它修改任何与其他进程共享的虚拟页面，除非所有的共享者都显式地允许它这么做(通过调用明确的进程间通信系统调用)。

为了防止内存被非法访问，会在 `PTE` 上添加一些额外的许可位来控制对一个虚拟页面内容的访问。

- `sup`，进程是否运行在内核模式下才能访问。
- `read`、`write`，读写控制访问。

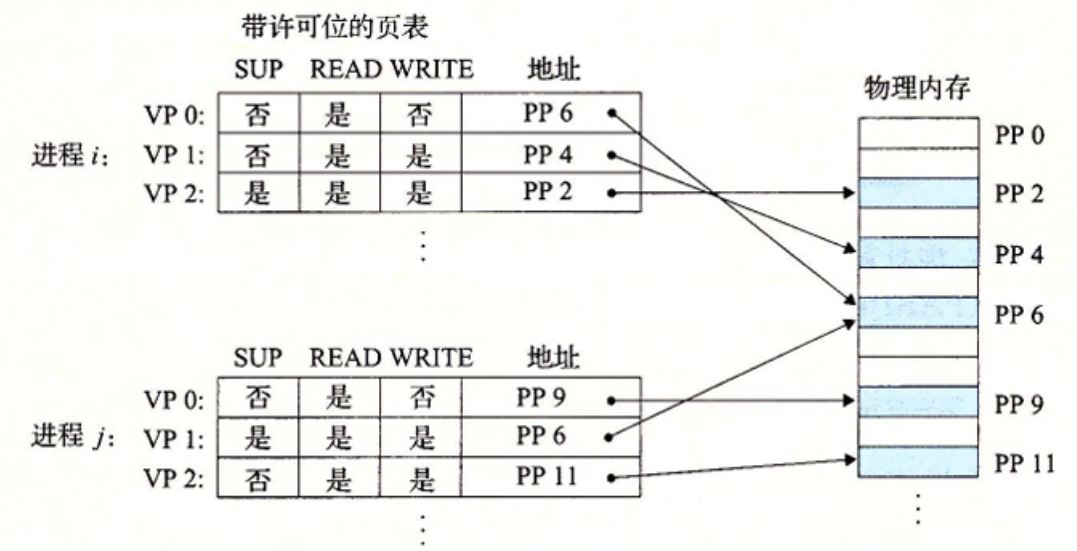

图 9-10　用虚拟内存来提供页面级的内存保护

地址翻译

地址翻译就是把一个`N`元素虚拟地址空间(VAS)中的元素映射到`M`元素物理地址空间(PAS)中的元素上

简单来说就是将虚拟地址转化为物理地址

- CPU中有一个页表基地址寄存器(Page Table Base Register, PTBR)指向当前页表，用于快速定位。
- n 位的虚拟地址划分为 p 位的虚拟地址偏移 `VPO` 和($n-p$)位的虚拟页号VPN。
- m 位物理地址划分为 p 位的物理地址偏移 `PPO` 和($m-p$)位的物理页号PPN。

因为虚拟页和物理页大小都是 P 字节，所以 `VPO` 和 `PPO` 是相同的。`MMU` 利用 `VPN` 来选择 `PTE` ，进而得到 `PPN` ，由于 `VPO` 与 `PPO` 相同，因此将 `PPN` 与 `VPO` 串联起来就得到了物理地址。

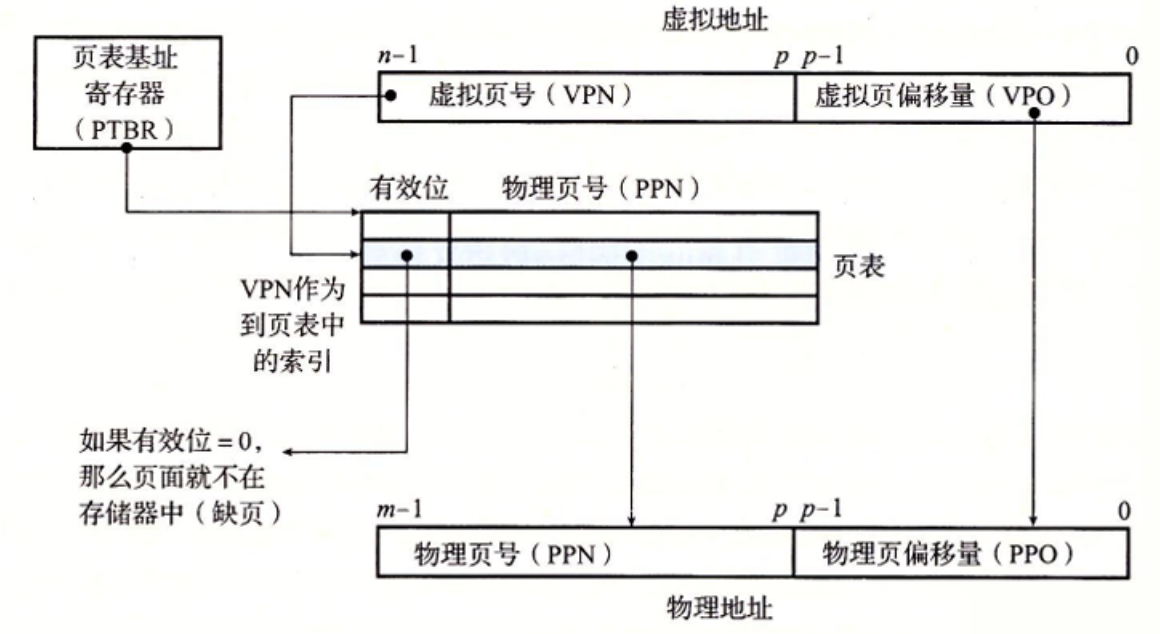

图 9-12　使用页表的地址翻译

为了能够更加直观的演示地址翻译的全过程, 我们举下面的例子 :

我们约定入下 :

- 虚拟地址是 14 位长的 $n=14$, 物理地址是 12 位长的 $m=12$

- 页面大小是 64 个字节的 $P = 64$

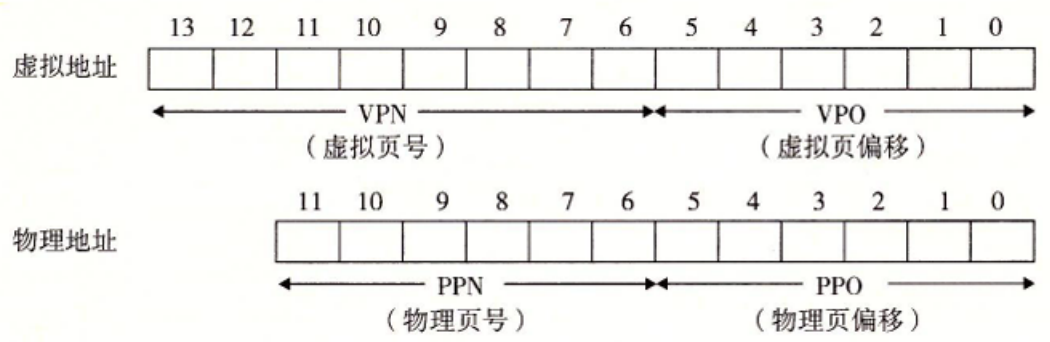

图 9-19 小内存系统的寻址。假设 14 位的虚拟地址($n=14$)，12 位的物理地址($m=12$)和 64 字节的页面($P=64$)

- 我们要翻译的虚拟地址是 $0\text{x}0\text{E}1\text{A}$

具体的步骤如下：

页命中情况

1. 确定 `VPN` `PPN` `VPO` `PPO` 的位数

   > 因为一个虚拟页(物理页) 的大小是 64 个字节, 转化为二进制为 2^8 , 所以虚拟页偏移量(VPO) 和 物理页偏移量 (PPO) 是有 8 位的, 有因为虚拟地址总共 14 位, 所以虚拟页号一共有 6 位, 凑成 2 的进制倍为 8 位
   >
   > 同理, 物理页号一共有 4 位

2. 提取出 `VPN` 和 `VPO`

   > 根据我们算出的位数提取出来即可, 提取结果如下：
   >
   > $\text{VPN} = 0\text{E} \qquad \text{VPO} = \text{PPO} = 1\text{A}$

3. 根据 CPU 中的页表基址寄存器查看页表信息, 并找到对应的 `VPN`

   > | VPN | PPN | 有效位 | VPN | PPN | 有效位 |
   > |---|---|---|---|---|---|
   > | 00 | 28 | 1 | 08 | 13 | 1 |
   > | 01 | — | 0 | 09 | 17 | 1 |
   > | 02 | 33 | 1 | 0A | 09 | 1 |
   > | 03 | 02 | 1 | 0B | – | 0 |
   > | 04 | — | 0 | 0C | – | 0 |
   > | 05 | 16 | 0 | 0D | 2D | 1 |
   > | 06 | — | 0 | 0E | 11 | 1 |
   > | 07 | — | 0 | 0F | 0D | 1 |
   >
   > b) 页表：只展示了前 16 个 PTE
   >
   > 发现 0E 对应的 `PPN` 为 11 , 同时有效位为 1 , 说明该虚拟页已分配并且已缓存
   >
   > 我们将两个部分组装在一起, 得到物理地址 $0\text{x}111\text{A}$

这是最简单并且最顺利的情况, 但是我们也要考虑缺页的处理情况

缺页情况

如果这个时候我们需要翻译的虚拟地址是 $0x0504$

那么当我们进行到第三步的时候, 会发现页表对应的有效位不为 1 , 但是内容也不是 $null$, 说明这个虚拟页是已分配但是没有在缓存中的, 这个虚拟页所对应的虚拟地址是 $0x1604$, 于是 `MMU` 就会触发一个缺页异常, 系统进入内核态然后调用缺页异常处理程序 , 缺页异常处理程序进入缓存, 进行组选择, 行匹配, 将对应的物理页内容放到缓存中后, 修改页表中的有效位为 1

之后 CPU 再次向 `MMU` 发送对于虚拟地址 $0x0504$ 的翻译请求, 这次页命中返回

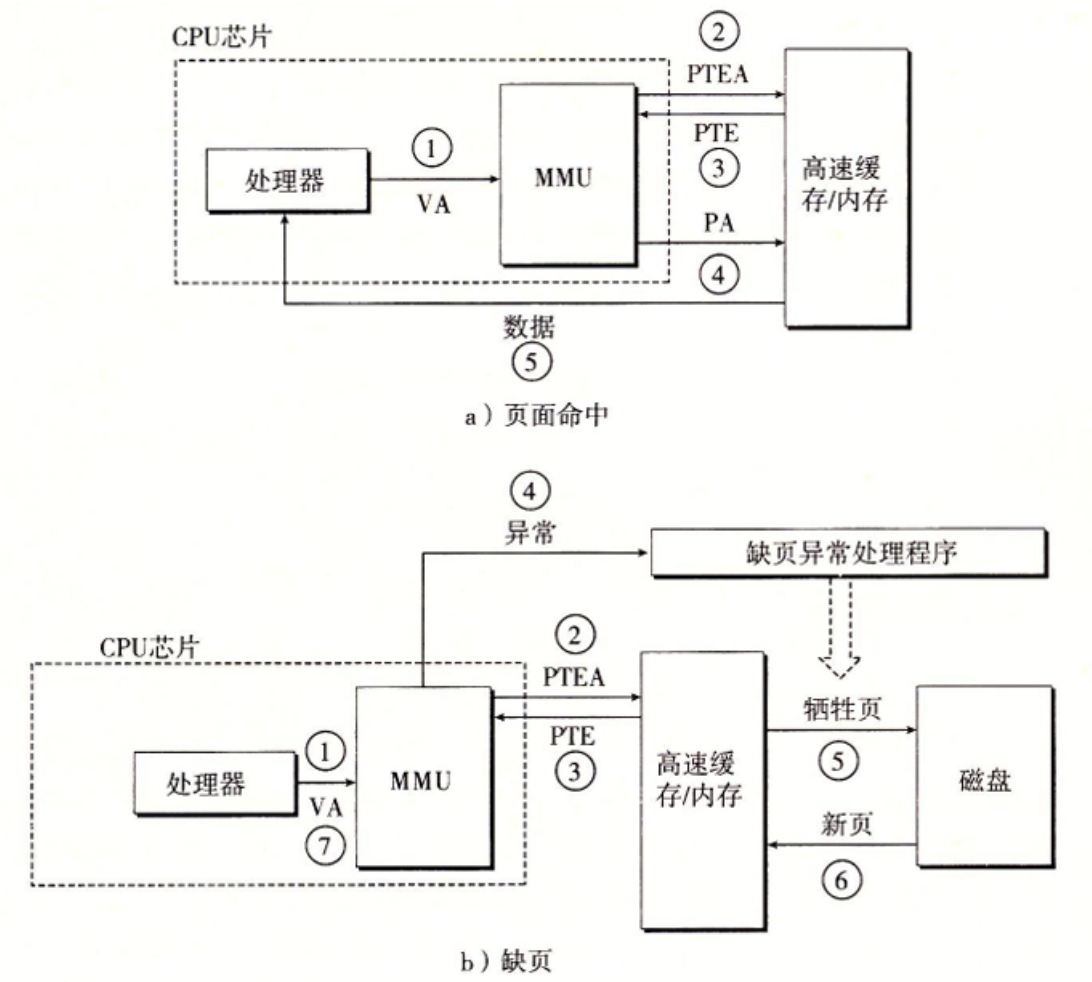

图 9-13 页面命中和缺页的操作图(VA：虚拟地址。PTEA：页表条目地址。PTE：页表条目。PA：物理地址)

使用 TLB 加速地址翻译

我们发现, 如果进行大量的地址翻译的话, 每次都需要从 CPU中的页表基址寄存器中取出起始地址, 然后再到内存中读取对应的内容, 这会十分的浪费时间, 如果需要的表项在 L1 cache 中还好, 如果不命中向下级请求的话开销非常大, 于是考虑用一个专门的缓存来缓存表项, 这个缓存就叫 TLB, 也叫快表

TLB 也分命中和不命中, 具体的细节和我们之前讲的 存储器体系结构相似, 也就不做赘述了

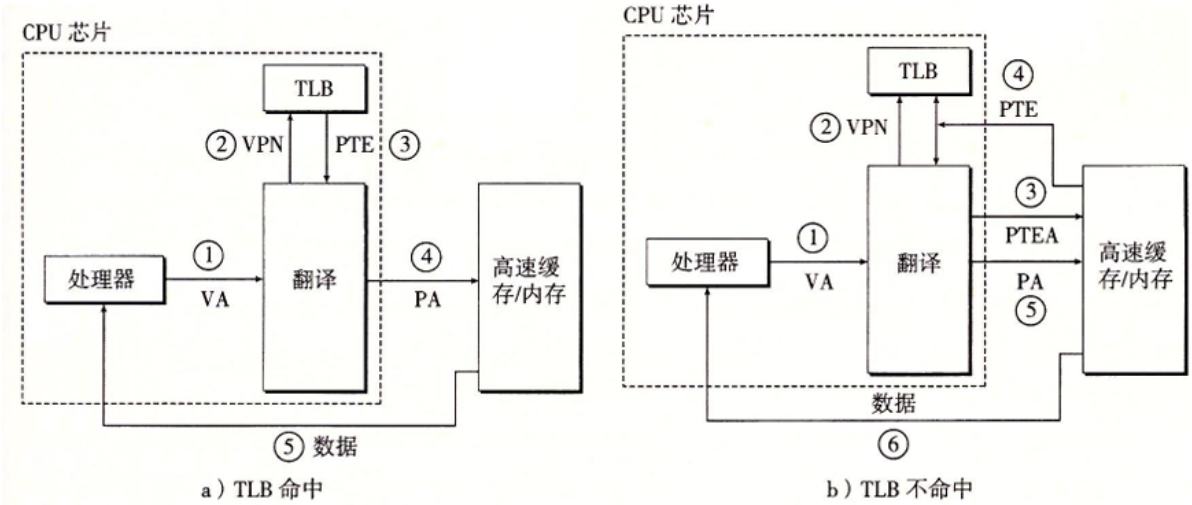

图 9-16　TLB 命中和不命中的操作图

多级页表

我们一开始讲过, 在内存中开辟一段连续的地址空间十分的困难, 但是值得注意的是, 我们的页表就需要一段连续的地址空间来存储信息, 如果我们的地址空间过多的话, 那么存储页表本身就是一件不可能的事情

所以我们采用分级的方式去存储页表, 我们将一个虚拟地址组织成下面的形式

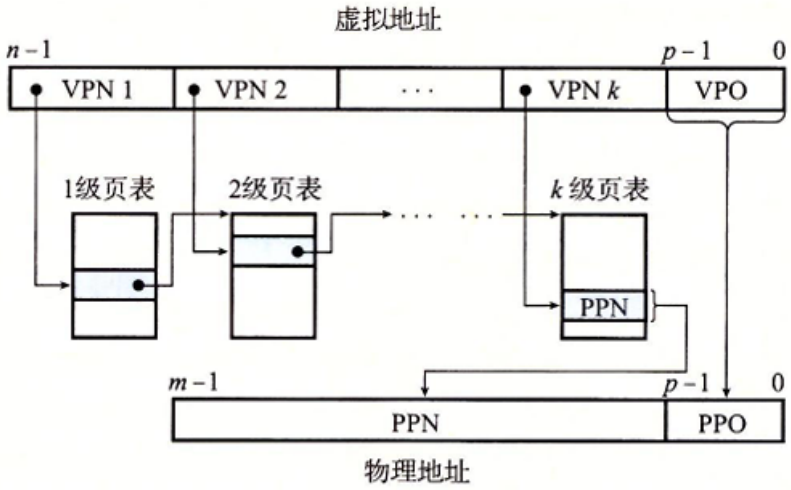

图 9-18　使用 k 级页表的地址翻译

同时一级页表也不再存储物理页的信息, 而是存储二级页表的地址

于是整体的结构为 :

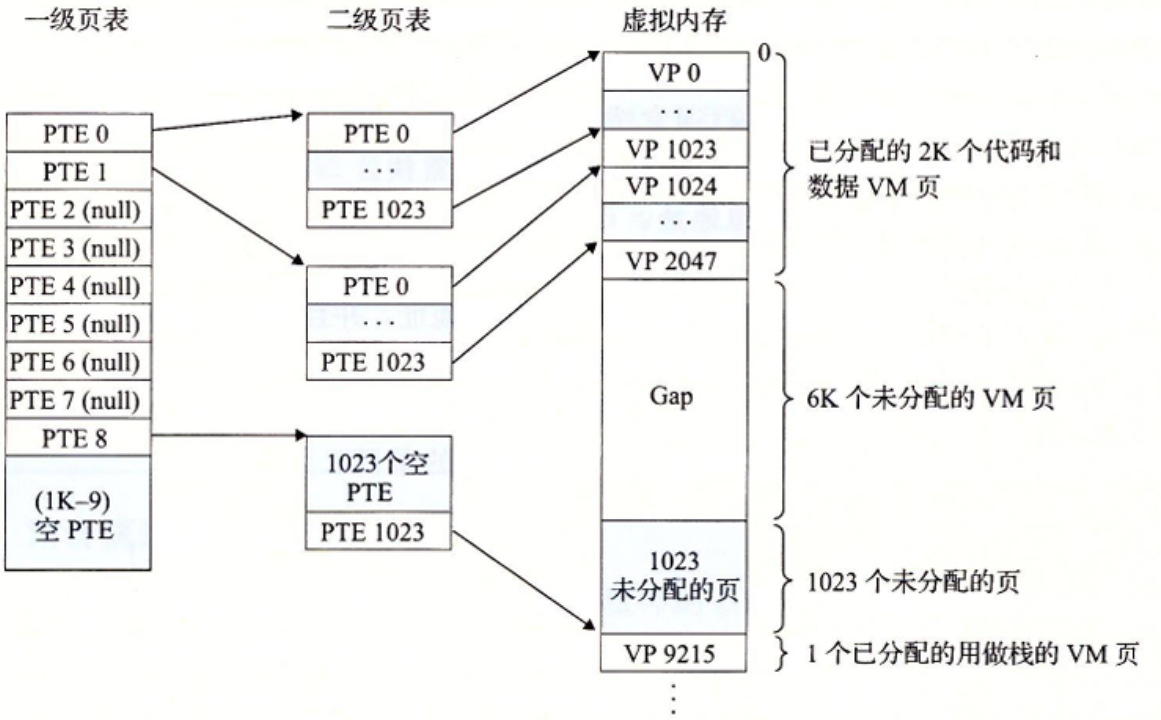

图 9-17　一个两级页表层次结构。注意地址是从上往下增加的

Core i7 的地址翻译

其实这一点和我们之前讲的都差不多, 主要是要自己看书, 页码为 576 ～ 579

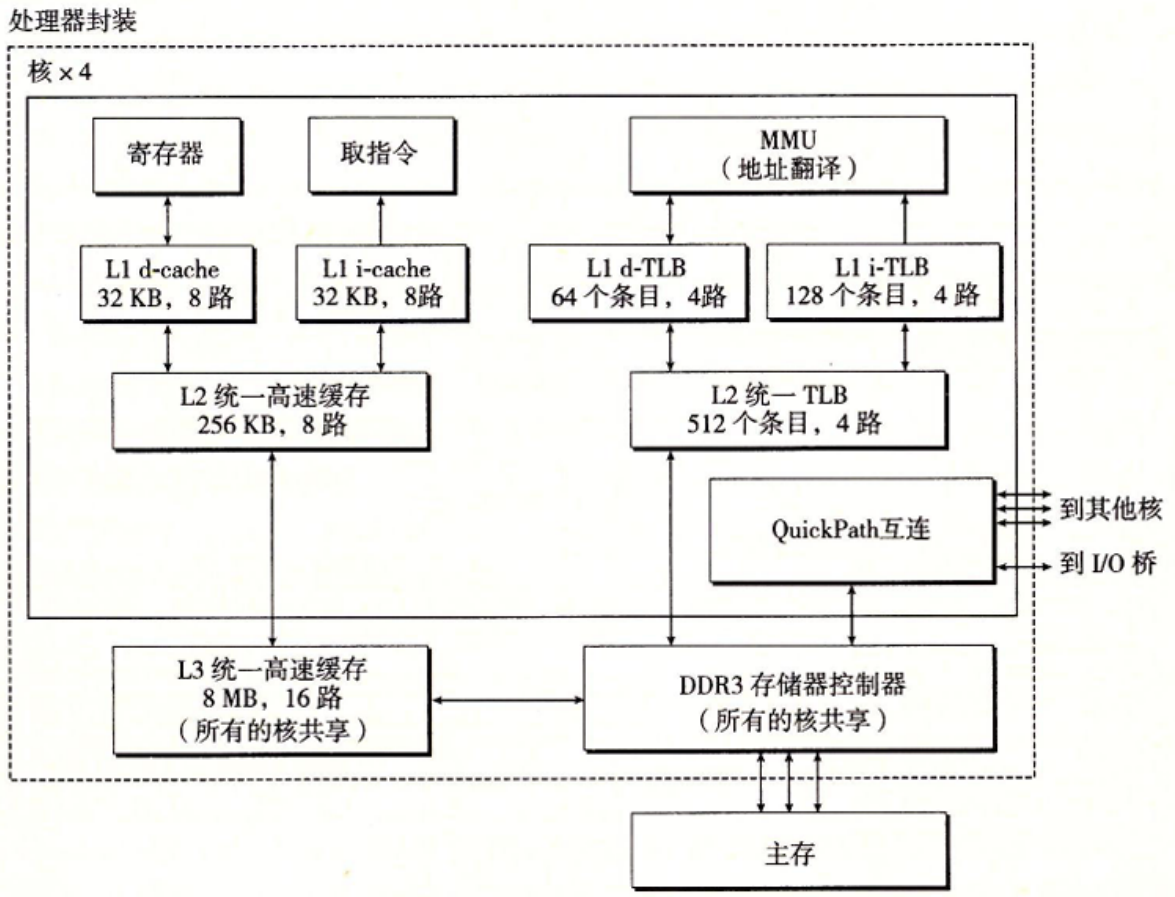

图 9-21　Core i7 的内存系统

Linux 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

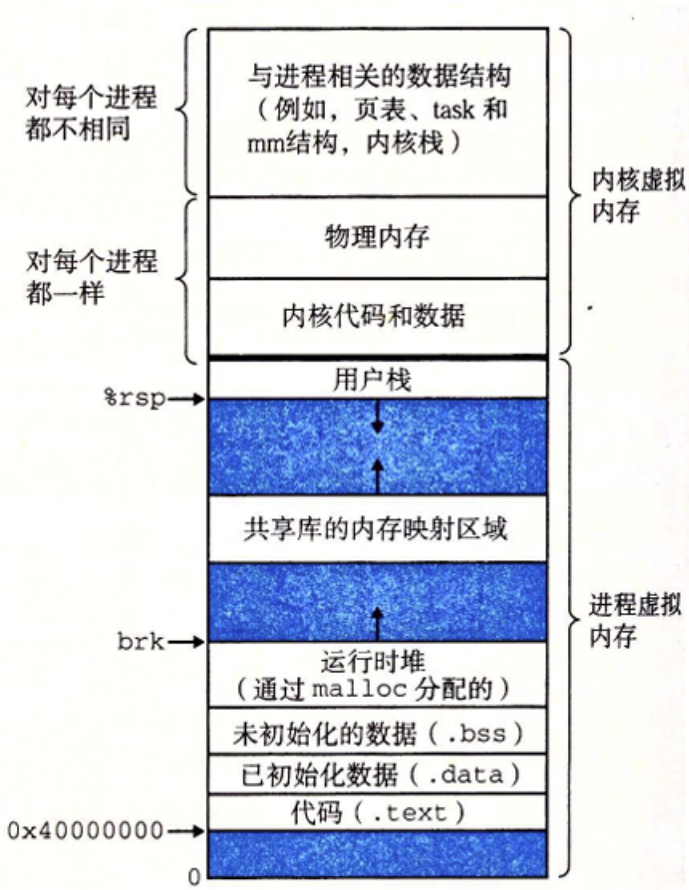

图 9-26　一个 Linux 进程的虚拟内存

可以看到, 我们可以将上述的图划分为两种状态, 上面的叫内核态, 下面的叫用户态

在内核态中 :

- 内核虚拟内存的某些区域被映射到所有进程共享的物理页面。例如，每个进程共享内核的代码和全局数据结构。Linux也将一组连续的虚拟页面(大小等于DRAM的总量)映射到相应的一组连续的物理页面，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。
- 内核虚拟内存的其他区域包含每个进程都不相同的数据。例如，页表、内核在进程上下文中执行代码时使用的栈、以及记录虚拟地址空间当前组织的各种数据结构。

Linux虚拟内存区域

Linux将虚拟内存组织成一些区域(也叫做段)的集合，一个区域就是已经存在的(已分配的)虚拟内存的连续片，这些页是以某种方式相关联的。如代码段、数据段、堆、共享库段和用户区都是不同是区域。

- 只要是存在的虚拟页就保存在某个区域中，不属于某个区域的虚拟页面是不存在的，并且不能被进程所引用。
- 区域的存在允许虚拟地址空间有间隙。
- 内核不记录不存在的虚拟页，而这样的页也不占用内存、磁盘或者内核本身中的任何额外资源，由此节省空间。

上面这段话说人话就是 : 虚拟内存中的内容都是一段一段出现的, 像我们之前在 链接 中讲到的, 一些有着相同类别的数据被组织在了一起, 像 `.bss` , `.text` 等, 在一个段内, 元素的访问权限都是一定的, 所以内核就会记录这些段的开头, 结尾, 访问权限等等一系列信息, 这些信息都存储在内核态中.

我们现在就来介绍一下, 在内核中是如何存储这些段的信息的 :

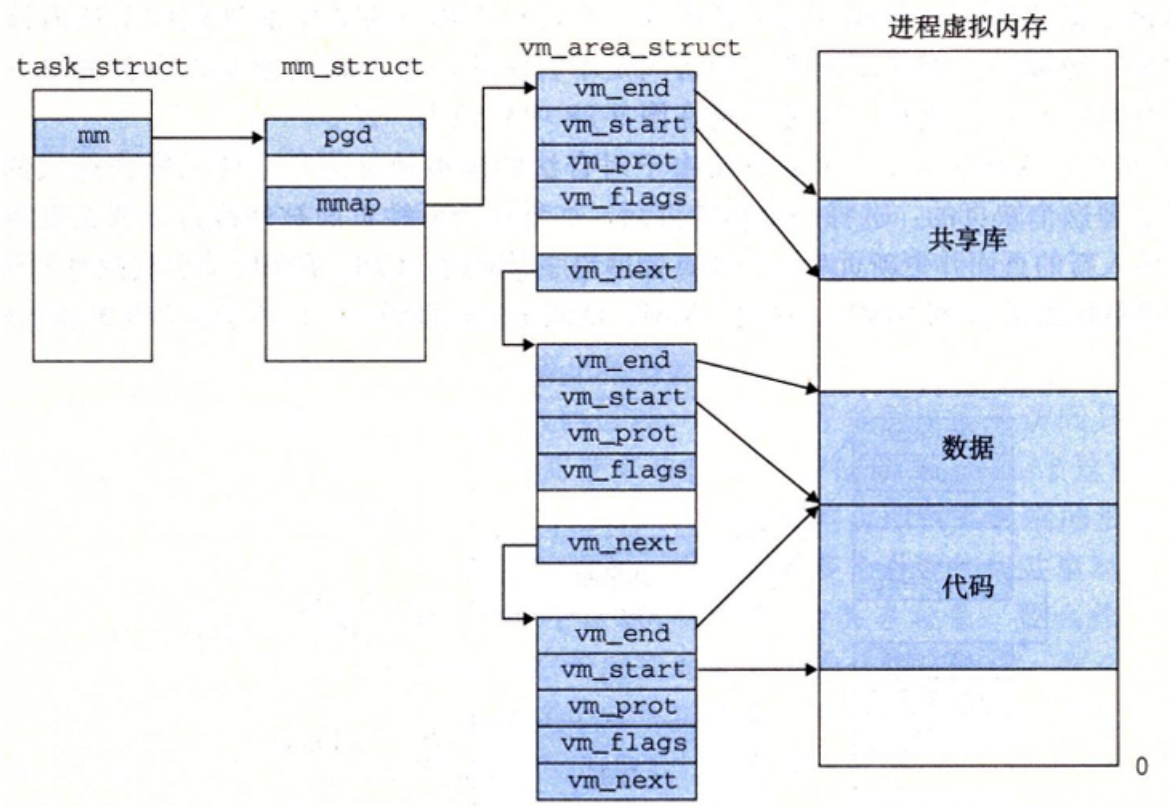

图 9-27　Linux 是如何组织虚拟内存的

在我们最开始展示的 Linux 虚拟内存组织结构图中, 内核态上有一个 `与进程相关数据结构` 部分, 这个数据结构就是 `task_struct`, 我们关系 `task_struct` 中的 `mm` 项. `mm` 项指向了一个 `mm_struct` 的数据结构, 这个数据结构中我们只关心两项

- `pgd` : 指向第一级页表 (页全局目录) 的基址
- `mmap` : 指向一个叫 `vm_area_struct` 的数据结构

`vm_area_struct` 可以看成是一个链表结构, 每一个节点存储着一个段的相关信息

- `vm_start`: 指向这个区域的起始处
- `vm_end`: 指向这个区域的结束处
- `vm_port`: 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
- `vm_next`: 指向链表中下一个区域结构

Linux 的缺页异常处理

当 `MMU` 翻译一个虚拟地址 A 时发生发生缺页异常，该异常使控制转移到内核的缺页处理程序，程序执行如下步骤:

- 判断 A 是否合法，即 A 是否在某个区域内。缺页处理程序搜索区域结构的链表，将虚拟地址与每个区域结构的 `vm_start` 和 `vm_end` 进行比较，由此判断虚拟地址是否合法，即是否在某个区域结构定义的区域内。若不合法，则缺页处理程序触发段错误，终止进程。
- 判断进程是否有读、写、执行这个区域内页面的权限，即内存访问是否合法。若不合法，则触发一个保护异常，终止进程。
- 若是通过了上述两步，则说明该缺页是对一个合法地址的合法操作导致的，由此则可以处理缺页。选择一个牺牲页，如果牺牲页被修改过，那么把它交换出去，换入新的页面并更新页表。缺页处理程序返回时，CPU重新启动引起缺页的指令，这条指令将再次发送 A 到 `MMU`。

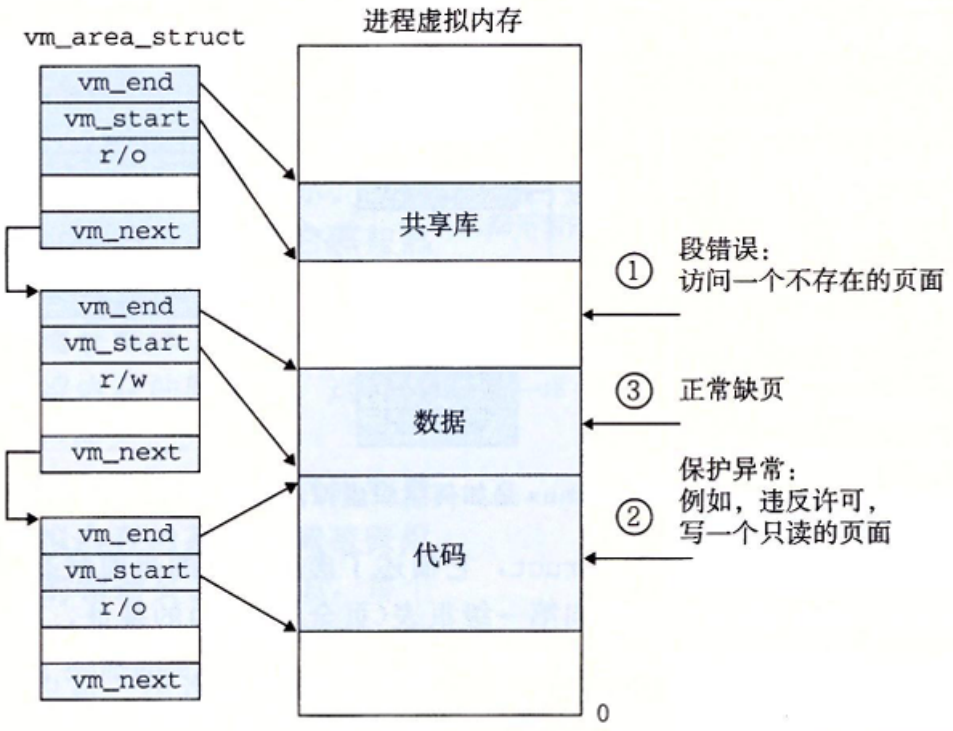

图 9-28　Linux 缺页处理

内存映射

虚拟内存区域是和磁盘中的文件对应的，Linux通过将一个虚拟内存区域与一个磁盘上的对象(也就是文件)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)。虚拟内存区域可以映射到以下两类对象：

- Linux文件系统中的普通文件

  > 一个区域可以映射到一个普通磁盘文件的连续部分，比如一个可执行文件。
  >
  > 文件区被分成页大小的片，每一片包含一个虚拟页面的初始内容。因为按需进行页面调度，所以这些虚拟页没有实际交换进入物理内存，直到CPU第一次引用虚拟页面(发射一个虚拟地址，落在这个页面的地址空间范围之内)。

- 匿名文件

  > 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，内容全是二进制零。
  >
  > CPU第一次引用这样一个区域的虚拟内存时，内核就在物理内存中找一个合适的牺牲页面，如果页面被修改过就将页面换出来，用二进制零覆盖牺牲页面并更新页表，并将这个页面标记为留在内存中。在磁盘和内存间没有实际的数据传输，因此映射到匿名文件中的页叫做请求二进制零的页 (demand-zero page)。
  >
  > 可以类比我们在 链接 中所讲的 `.bss`，就只是占位符作用

无论哪种情况，一旦一个虚拟页面被初始化了，它就在一个由内核维护的专门的**交换文件**(swap file)之间换来换去。交换文件也叫交换空间、交换区域。注意该空间限制当前运行的进程能够分配的虚拟页面的总数。

再看共享对象

操作系统为每个进程提供私有的虚拟地址空间，可以免受其他进程读写的干扰。但对于每个进程都要访问的相同的只读代码区域，如果每个进程在物理内存中保存一份副本，那就是极大的浪费。因此还是希望进程能够共享某些对象，即多个进程共享内存中的同一份资源

一个对象可以被映射到虚拟内存的一个区域，要么作为**共享对象**，要么作为**私有对象**。一个映射到共享对象的虚拟内存区域叫做**共享区域**，类似的，也有**私有区域**。

我们简单解释一下共享区域和私有区域的区别 :

- 在共享区域中 : 如果一个进程修改了这部分内容, 那么是会真正修改这部分的物理地址内容的, 只不过 CPU 不保证什么时候将修改的内容写入, 也就是说 : 一个进程修改内容, 其他进程是可以看见修改的
- 在私有区域中 : 如果一个进程修改了这个部分的内容, 对于修改的进程来说确确实实是修改了的, 但是对其他共享此处内存的进程来说是没有变化的

Linux 内存映射之文件映射 - 知乎 (zhihu.com)

下面的图展示了共享区域的映射

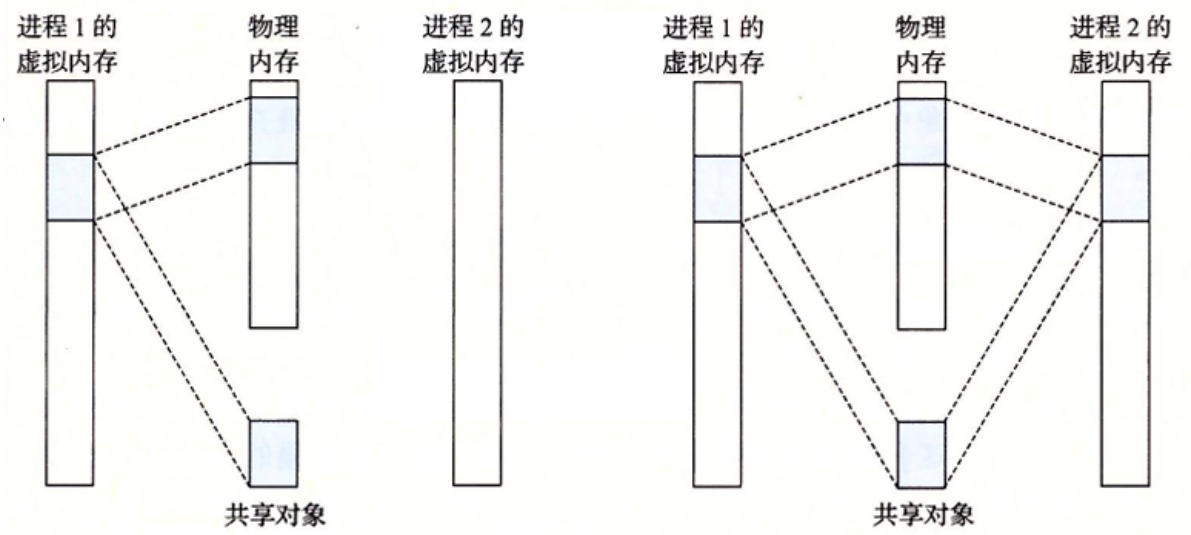

图 9-29　一个共享对象(注意，物理页面不一定是连续的)

私有对象使用一种叫做**写时复制**的巧妙技术被映射到虚拟内存中。对于下图有两个说明:

- 在物理内存中保存私有对象的一个副本，只要没有进程试图写私有区域，进程就可以一直共享物理内存中对象的一个单独副本，且每个进程私有区域的页表条目都被标记为只读，区域结构被标记为私有写时复制。
- 若有进程试图写私有区域的某个页面，会触发一个保护故障，它会在内存中创建这个被写页面的新副本，然后更新页表条目指向新副本，并恢复这个页面的可写权限。

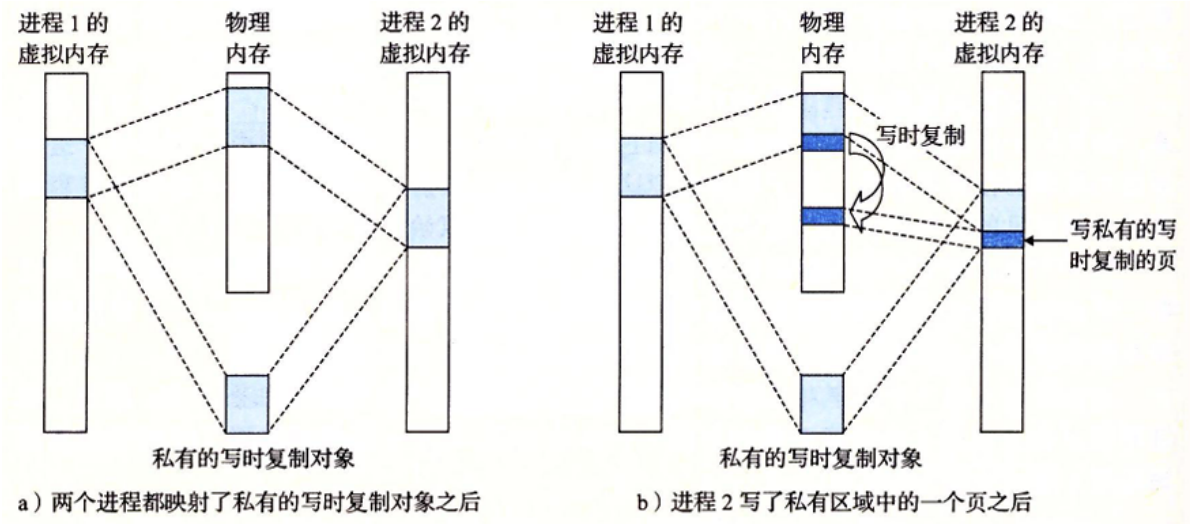

图 9-30　一个私有的写时复制对象

再看 fork 函数

Linux下可以使用 `fork` 函数创建新的进程，显然创建的新进程带有自己独立的虚拟地址空间。在当前进程调用 `fork` 函数时，内核为新进程创建各种数据结构，并为其分配唯一的进程ID。为给新进程创建虚拟内存，创建当前进程的 `mm_struct` 、区域结构和页表的副本。两个进程中的每个页面都标记为只读，**两个进程中每个区域结构都标记为私有写时复制。**

当fork在新进程中返回时，新进程现在的虚拟内存刚好和调用fork时存在的虚拟内存相同。当这两个进程中某个进行写操作时，写时复制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

再看 execve 函数

`execve` 函数在当前进程中加载、运行包含在可执行文件 `a.out` 中的程序，用 `a.out` 有效代替当前程序，步骤如下：

- 删除当前进程虚拟地址的用户部分已存在的区域结构。
- 映射私有区域，如代码区、数据区分别映射 `a.out` 文件的 `.text` 、 `.data` 区等。
- 映射共享区域，如 `a.out` 与 `libc.so` 库链接，则将库映射到用户空间的共享区域。
- 设置当前进程上下文的程序计数器 (PC)，使之指向代码区域的入口

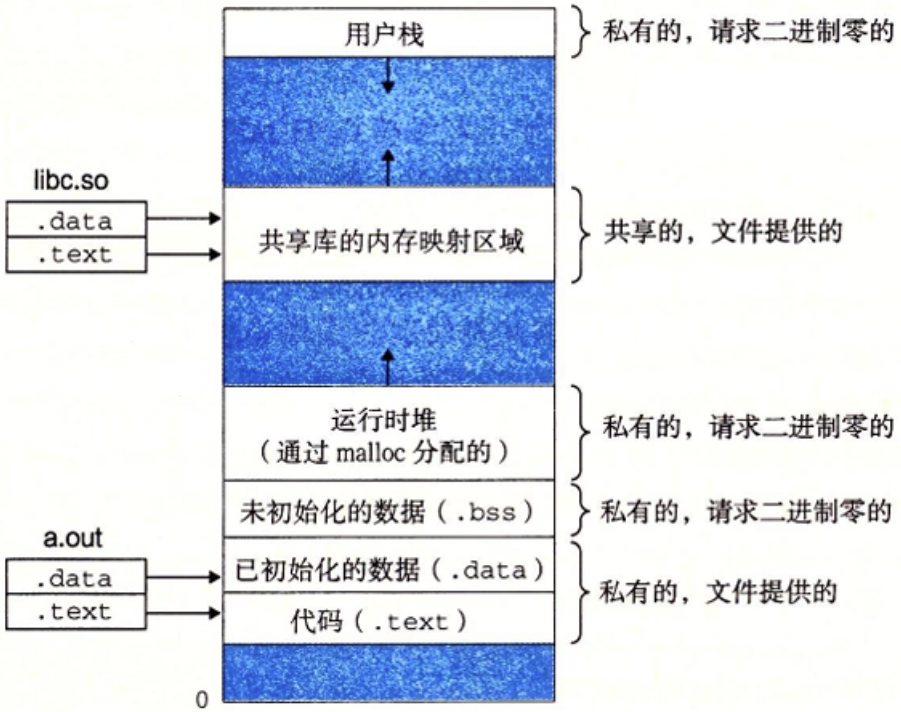

图 9-31　加载器是如何映射用户地址空间的区域的

使用 mmap 函数进行用户级内存映射

Linux进程可以使用 `mmap` 函数来创建新的虚拟内存区域，并将对象映射到这些区域中。

```
#include <unistd.h>
#include <sys/mman.h>
void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t
offset);
```

返回 : 如果成功返回该映射区域的指针, 否则返回 MAP_FAILED(-1)

从fd指定磁盘文件的offset处，映射len个字节到一个新创建的虚拟内存区域，该区域从地址stat处开始。

这个 fd 是可以配合 `Open` 函数使用的

- start：虚拟内存的起始地址，通常定义为NULL
- prot：虚拟内存区域的访问权限

  > PROT_READ(可读)
  >
  > PROT_WRITE(可写)
  >
  > PROT_EXEC(可执行)
  >
  > PROT_NONE(不能被访问)

- flags：被映射对象的类型

  > MAP_ANON(匿名对象)
  >
  > MAP_PRIVATE(私有的写时复制对象)
  >
  > MAP_SHARED(共享对象)

- 返回值：指向映射区域开始处的指针

同时还有一个删除该虚拟内存区域的函数 `munmap`

```
#include <unistd.h>
#include <sys/mman.h>
int munmap(void* start, size_t length);
```

成功返回 0，失败返回 -1

`mumap` 函数删除虚拟地址从 start 开始, 接下来 length 字节组成的区域

动态内存分配