文章已收录我的仓库：Java学习笔记与免费书籍分享

分段、分页

引言

什么是碎片？

碎片分为内部碎片与外部碎片，都是指浪费而不能使用的空间。

内部碎片是指**已分配但未被使用的地址空间。**例如在64位空间内，你只使用 7 字节但由于内存对齐不得不为你分配8字节空间，这就产生了1字节内部碎片。

外部碎片是指**未分配且未使用的地址空间。**例如，你申请4字节的Int类型，再申请8字节的long类型，为了内存对齐，其中4字节无法装入8字节类型，这就产生了4字节的外部碎片，如下图所示。

内部碎片是已被分配的空间，是操作系统不可利用的空间；外部碎片是未被分配的，是可分配的，但该空间过小(碎片的含义)无法装入资源，导致不可利用，但外部碎片是可解决的，可以将多个外部碎片紧凑成一个大的空闲空间，但这需要大量成本。

段式模型的前身：基址加界限寄存器(动态重定位)

要想理解分段与分页，必须先谈谈早期的虚拟内存模型。

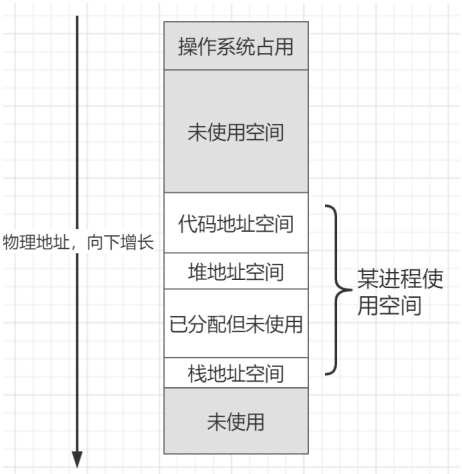
在经历了纯物理地址后，科学家们期望解决这种内存模型难以统一的问题，于是虚拟内存技术孕育而生，但困扰科学家们的是，如何将虚拟地址转换成物理地址。

早期的科学家们很容易的想到将整个程序作为一个整体，并为每个进程分配一个基址寄存器和界限寄存器**，基址寄存器存放该虚拟地址在实际物理地址的起点，而界限寄存器则用以判定程序是否访问非法地址**。

通过这种方式，实际的地址很好计算:

$$实际地址 = 虚拟地址 + 基址$$

但是，这种方式仍然将进程的全部地址空间加载内存中，虽然解决了地址翻译问题，但仍然产生了大量的内部碎片，如下图中该进程栈堆区很小，于是在栈堆区之间产生了**内部碎片**。



从图中可以看出，如果我们将整个地址空间放入物理内存，那么栈和堆之间的空间并没有被进程使用，却依然占用了实际的物理内存。因此，简单的通过基址寄存器和界限寄存器实现的虚拟内存很浪费。

另外，我们必须要确保内存足够放得下进程的虚拟地址空间，但通常主存成本是比较昂贵的，不如磁盘廉价，这种方式通常不支持大的虚拟地址，如果剩余物理内存无法提供连续区域来放置完整的地址空间，进程便无法运行。例如现在32位的进程空间通常是4GB，主存根本就装不下几个进程。

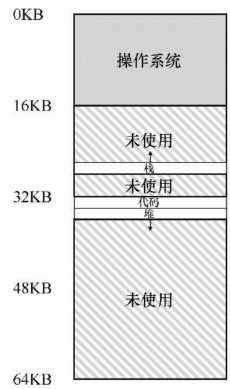
关键问题是：**怎样支持大虚拟地址空间**，同时栈和堆之间（可能）有大量空闲空间？在之前的例子里，地址空间非常小，所以这种浪费并不明显。但设想一个32位（4GB）的地址空间，通常的程序只会使用几兆的内存，但却需要一次性的将整个地址空间都放在内存中。

我们需要更复杂的机制以利用物理内存，避免内部碎片，早期的科学家们想出了分段这种思想。

分段式管理

分段思想

分段思想其实就是将基址加界限的概念泛化，现在，我们**为代码、堆和栈段分别维护一个段基址加段界限寄存器**，这样我们不必要每次都强制的装入整个进程空间，**每个基址寄存器存放该段在物理地址的实际空间，界限寄存器仍然用于保护地址空间。**



现在对于程序未使用的**段空间**，我们没必要为其分配了，因为每个段之间的耦合度降低了，一个空闲的段并不会影响其他的段，等到需要用时，段可以动态生长，但一旦超过段长，仍然会触发异常。

操作系统也可以离散的分配空间，例如堆和栈不必是连续的空间（基质寄存器不同），即各个段之间物理内存中的地址不一定是连续的，这也能大大的提高对物理地址的利用率。

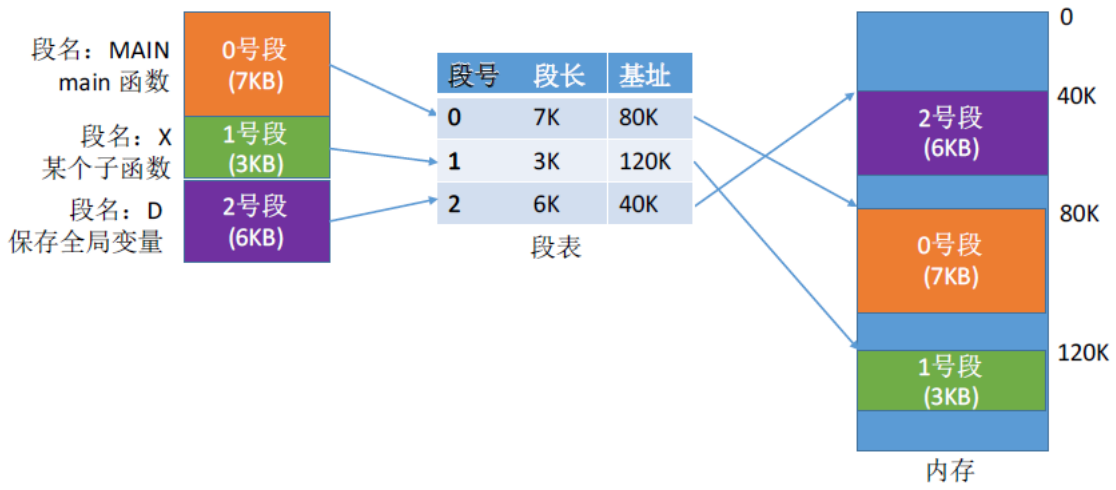
从某种层面上说，由于段是动态生长的，未使用的空间将不会进行分配，此时可以认为段不会产生内部碎片。

但是，剩下的空间也必须预留给这个段，以支持段的生长，如果将段长分的小一点，那么内部碎片最大不超过这个段的大小，合理的设置段大小可以有效避免内部碎片。 当然，如果内存空间没有限制，操作系统可以动态的去修改段表中的段长。

此外，由物理地址不必是连续的，一些地址可以在运行时额外分配一个段，增添了灵活性。

分段地址转换

分段地址转换与基址加界限的思想大同小异，在分段思想中，程序可能具有多个段，操作系统通过一个段表来维护各段信息：



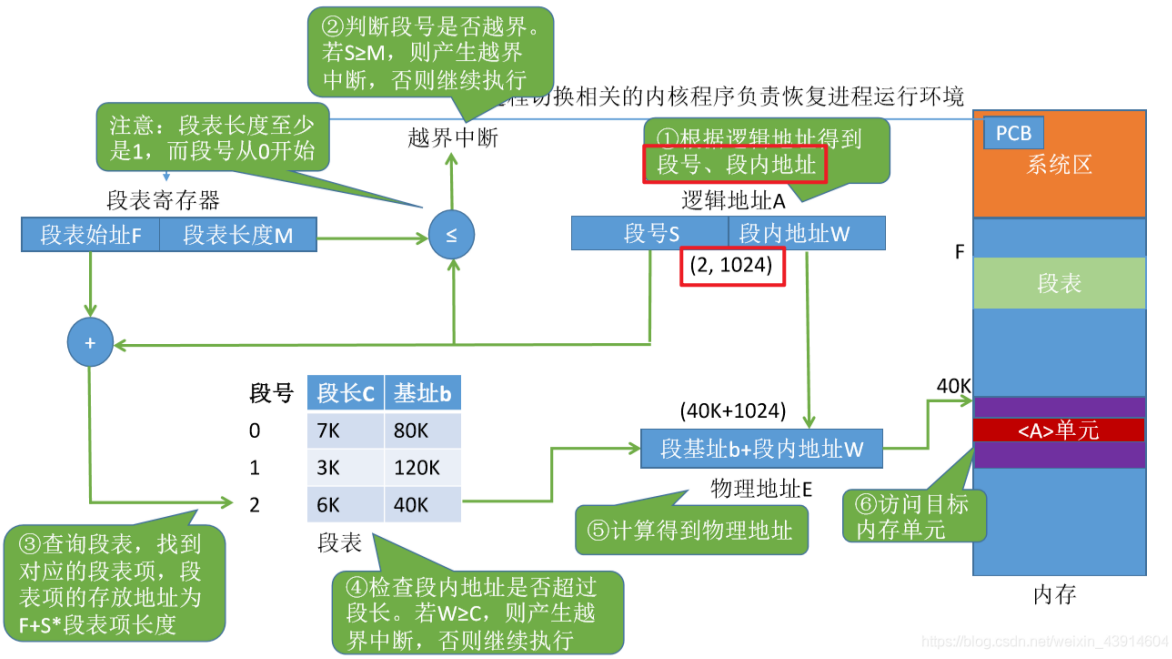
段表的地址是操作系统维护的，段表项主要维护段长和段基址，段基址指该段在物理内存中的起始地址，那么该段中的虚拟地址对于实际的地址即为 `段基址 + 段内偏移` **。**

分段系统的逻辑地址结构是由**段号**（段名）和**段内地址**（段内偏移量）所组成。

例如，若系统是按字节寻址，用32个二进制位表示逻辑地址，如果段号和段内地址各占16位，那么它的虚拟逻辑地址结构图如下所示。

31 16	15 0
段号	段内地址

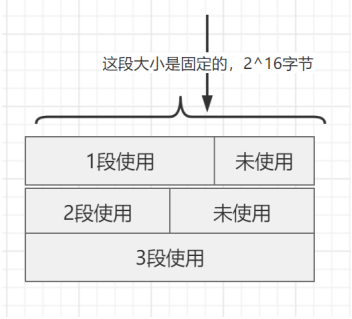
那么我们读取前16位作为段号，后16位作为段内偏移，**操作系统通过计算 addr = 段号 * 段表项大小 + 段表地址得出对应的段表项地址，通过查询该段表项得出段基址，通过计算 段基址 + 段内偏移得出物理地址。**



你可能发现了，在虚拟地址中，每个段的起始地址都是固定的，每个段的总大小都是固定的，其大小为：

$$size = 2^p字节，p = 段内地址的位数$$

如下图所示，注意展示的是虚拟地址的空间：



此外，栈地址是反向增长的，因此段表中必须维护一个比特位，描述是否为栈段。

段的另一个优点：很好的支持共享

随着分段机制的不断改进，系统设计人员很快意识到，通过再多一点的硬件支持，就能实现新的效率提升。具体来说，要节省内存，有时候在地址空间之间共享（share）某些内存段是有用的。尤其是，代码共享很常见，今天的系统仍然在使用。

为了支持共享，需要一些额外的硬件支持，这就是保护位（protection bit）。基本为每个段增加了几个位，标识程序是否能够读写该段，或执行其中的代码。通过将代码段标记为只读，同样的代码可以被多个进程共享，而不用担心破坏隔离。

为什么页不行？页的分配是随机的，一个页通常只有 4kb，例如如果需要共享的数据有 64kb，则需要分配16 个页，这些页可能是不连续的，因此无法很好的支持共享。如果将多个页组成一个逻辑页，这已经就是分段的思想了（段页式）。

因此我们常说，段式管理是符合用户逻辑的，是利于保护和共享的。

虚拟地址翻译太慢？

我们每次翻译一个虚拟地址都需要去找寻段表中的段表项，相当于多义词地址访问，这太慢了！解决方案是为计算机设置一个小型的硬件设备，将虚拟地址直接映射到物理地址，而不必再访问段表。这种设备称为转换检测缓冲区 （Translation Lookaside Buffer，TLB），有时又称为快表。

快表是一个小的高速缓存，现代操作系统无论是分段还是分页中都利用了这种软件技术，有关于快表地址翻译的问题我们将在专门针对地址翻译的文中讲解。

段的缺点：过多的外部碎片

分段可以避免产生内部碎片，但由于分段是离散的在主存内找到空闲的槽块并插入，问题是物理内存很快充满了许多空闲空间的小洞，因而很难分配给新的段，或扩大已有的段 —— **大量外部碎片。**

例如4kb的空间装入3kb的段，产生的1kb的空间无法在装入任何段，产生碎片的主要原因是因为分段使用的大小是不确定的。

当然前面也提到过，外部碎片可通过*紧凑*的方式以合成较大的空闲空间，但这需要大量成本，操作系统难以维护。

这种情况下，分页式管理应运而生。

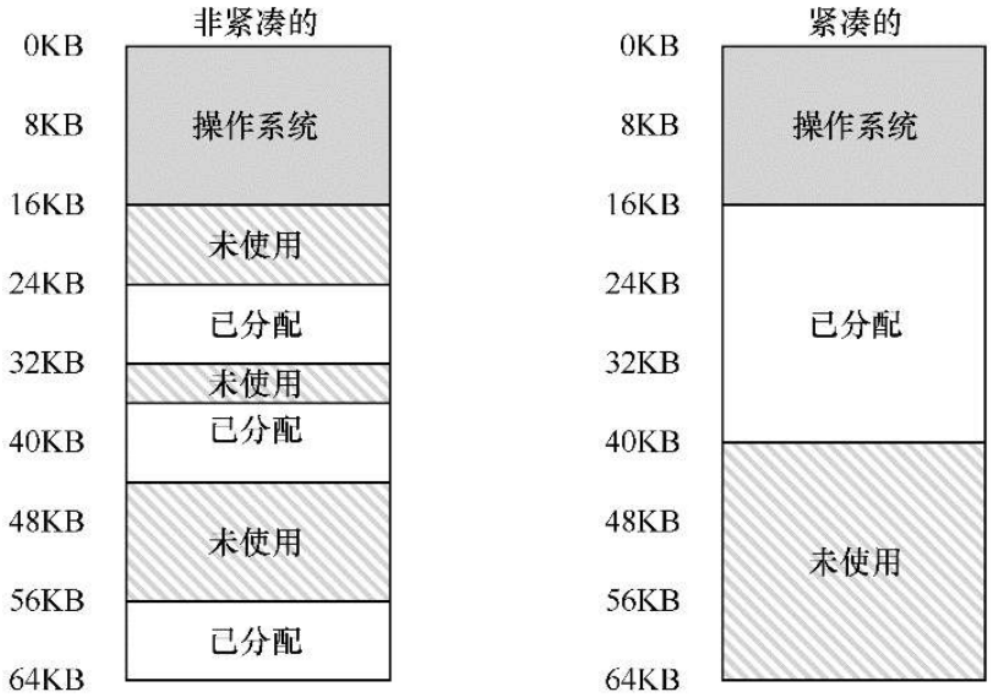


图16.3 非紧凑和紧凑的内存

分页式管理

分页思想

对于分段式的管理，一段时间后主存上将会遍布大大小小的外部碎片，操作系统难以进行维护，**分段的思想是将内存空间分割成不同长度的分片，由于长度不是固定的，产生外部碎片是必然的**，之前提到的将整个程序一起装入的方法虽然不会产生外部碎片，但会产生巨大的内部碎片，我们需要更细粒度的划分，以减少内部碎片的产生，解决这一问题的办法是**将空间分割成较小的、固定长度的分片，这就是分页式管理。**

分页式管理将程序资源划分为固定大小的页，将每一个虚拟页映射到物理页之中，由于每个页是固定大小的，操作系统可以整齐的分配物理内存空间，**避免产生了外部碎片**，例如一个页大小是4kb，而主存是40kb，操作系统稍加管理便能确保无论何时都能整齐的装入10个页面。

要注意到页在物理内存中也不是连续存在的，进程未使用的页也没必要为其分配内存，通过这种方式我们就解决了由分段产生大量外部碎片的问题，同时由于页较小，**只有在已使用的页才会产生少量的内存碎片**，这也是可以接受的，目前来看，分页是一个良好的解决办法。

分页地址转换

正如同分段一样，分页地址转换也需要`基址+页内偏移`来完成，在分段中采用段表来存储段基址，而在分页中则采用页表来存储页基址，页基址表示页在实际内存中的起始地址，那么实际的地址：

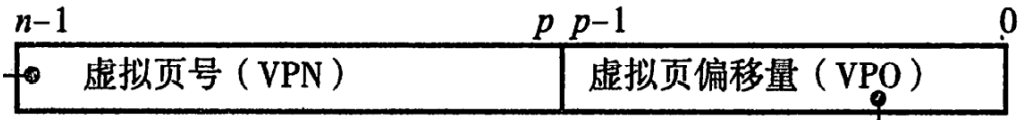
$$addr = 页基址 + 页内偏移$$

页表是由操作系统维护的，操作系统知道页表的起始位置，页表项的大小是固定的，在32位地址空间中，通常是8字节，这64比特中不仅存储了页基址，还存放着一些其他重要的数据，如：有效位、可读位、脏位等。

虚拟地址是由页表号 + 页内偏移组成的，这与分段中的虚拟地址类似，我们来进行一个简单的计算以得出32位程序中页表号所占用的位数，其中一个页表的大小通常是4kb，那么：

$$虚拟地址的总空间大小 = 2^{32} = 4GB \; 页表项的个数 = \frac{4GB}{4kb} = 2^{20} \; 4kb = 2^{12} byte$$

要能表示2^{20}个页表项，我们必须分配20位地址，剩余12位代表页内偏移，即下图中p=12，n=32，我们通常称虚拟页号为VPN，而称页偏移量为VPO，如下图所示：



为什么直接取VPO就代表了页偏移量？这也是很好理解的，因为：页偏移量 = 虚拟地址 - 页起始地址，而页起始地址其实是固定的，即当VPO位全为0时，为对应页的起始地址 ，此时 虚拟地址 - 页起始地址 即为VPO表示数值。

现在操作系统取出虚拟地址，我们设其为vAddr，便可以通过如下步骤翻译成物理地址：

1. 获取VPN与VPO，即`VPN = vAddr & 0XFFFFF000;VPO = vAddr & 0X00000FFF;`
2. 获取页表项地址，$页表项地址 = 页表起始地址 + VPN \times 页表大小$;
3. 从该页表项内取出页基址，即实际物理起始地址PPN(注意这里仅有20位，需要左移动12位才是真正的地址);
4. 将PPN与VPO连接起来，即$真实地址 = (PPN << 12) \ | \ VPO$

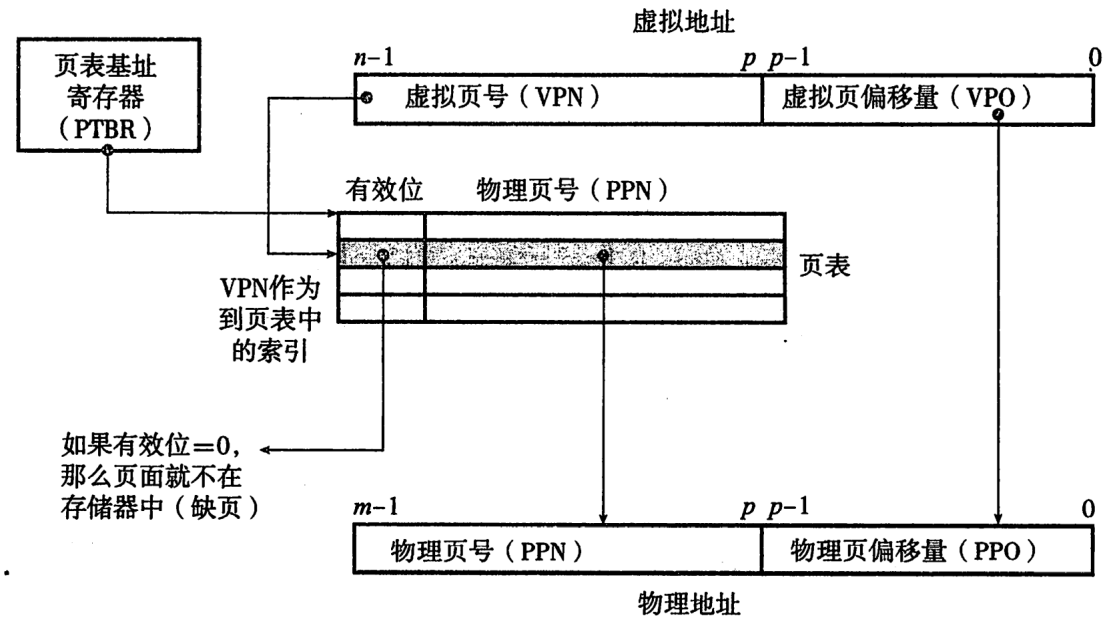
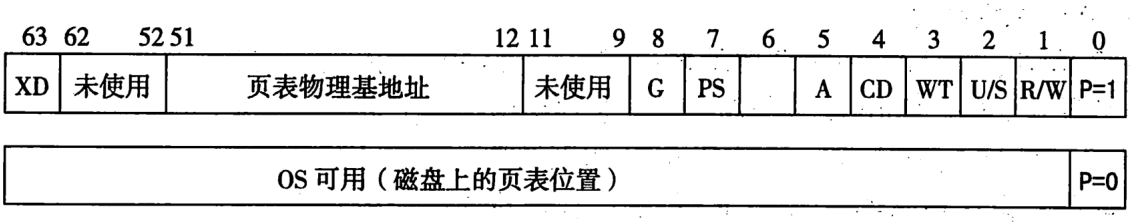


图 9-12　使用页表的地址翻译

实际的页表项还包含其他一些信息，如下图便是酷睿I7操作系统中的页表项：



字段	描述
P	子页表在物理存储器中（1），不在（0）
R/W	对于所有可访问页，只读或者读写访问权限
U/S	对于所有可访问页，用户或超级用户（内核）模式访问权限
WT	子页表的直写或写回缓存策略
CD	能 / 不能缓存子页表
A	引用位（由 MMU 在读和写时设置，由软件清除）
PS	页大小为 4 KB 或 4 MB（值对第一层 PTE 定义）
Base addr	子页表的物理基地址的最高 40 位
XD	能 / 不能从这个 PTE 可访问的所有页中取指令

图 9-23　第一级、第二级和第三级页表条目格式。每个条目引用一个 4 KB 子页表

分页的缺点：页表过大怎么办？

正如我们上面所计算的，对于32位操作系统而言，假定页大小为4kb，我们得出大概需要2^{20}个页表项，而每个页表项大小通常是8字节，这意味着页表的大小将是$2^{{20}} × 8byte = 4Mb$，这大的令人发指，然而大多数程序可能仅使用几mb的大小，页表的大小甚至比整个进程所需的所有资源还大，我们必须想办法解决这个问题，但前提是，我们仍然要支持进程的虚拟大地址空间，尽管进程可能用不上这么多。

你可能会想到，对于进程未使用的空间，操作系统不为其分配页表项以节省空间。

的确，这确实解决问题的办法，但关键在于，操作系统根本不可能做到真正意义上的不分配页表项，操作系统必须要确保每一个虚拟地址都具有意义。这句话也许优点拗口，让我们来看一个例子：

我们假设地址空间是三位的，前两位代表页号，后一位代表页偏移，那么进程虚拟地址共有8个，进程空间大小为8字节：

{ 000　页号为0, 偏移为0的地址 001　页号为0, 偏移为1的地址 010　页号为1, 偏移为0的地址 011　页号为1, 偏移

那么我们必须准备四个页表项，存放页0 ~ 3的物理起始地址，现在该进程没有使用页0与页1，我们假设操作系统没有维护0 ~ 1的页表项，现在对于页表而言，页号3是该页表的第一个偏移量，**这不对，页号3无法被正确访问！**

即时你想出某个办法使得页3能被正确翻译，但假设此时进程收到访问地址为000的指令呢？这是可能的，由程序在运行时生成的。现在整个系统都将陷入苦恼，根本没有任何关于地址000的信息。现在你可能理解了，操作系统必须要确保每一个虚拟地址都具有意义，当该虚拟地址未被使用时，也必须有一些信息来标识该地址未被进程使用，属于非法地址。

所以直接的方法是不管用的，解决这一问题的办法是在加一层抽象。

多级页表

在多级页表中，**上一级页表存放的是对应的下一级页表的起始地址，并至少存在一个有效位标识以标识下一级页表是否存在。**

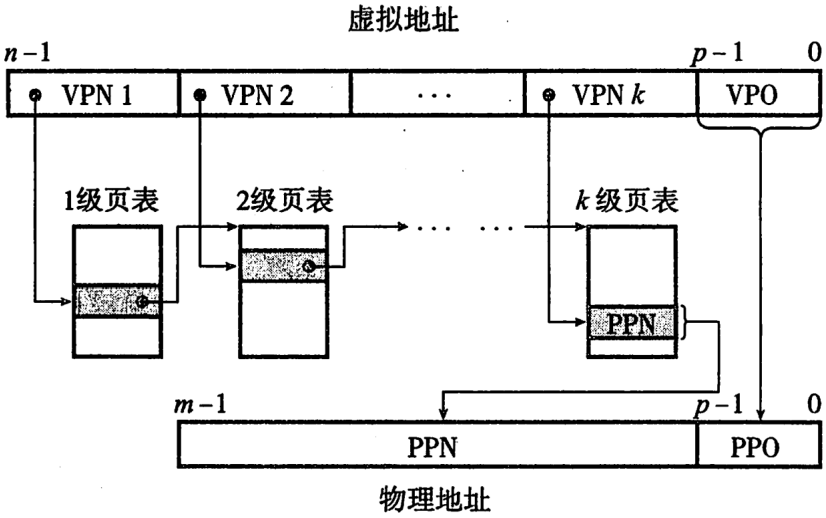


图 9-18　使用 k 级页表的地址翻译

看一个例子，仍然假设3位地址的操作系统，第一位表示一级页表，第二位表示二级页表，第三位表示页偏移：

{000　一级页表页号为0，二级页表页号为0，偏移为0的地址 001　一级页表页号为0，二级页表页号为0，偏移

我们有一个一级页表，一级页表有两项，一级页表项至少存在一个有效位，如果确实有效则还要保存下一级页表的起始地址。

我们仍然假设 000;001;010;011 这些地址进程未使用，现在假设进程访问地址010，MMU(地址翻译单元)取出一级页号 0，并访问一级页表偏移为0的页表项，此时操作系统发现该使用位设置为0(未使用)，则无须访问二级页表，并立即返回，告知进程该地址非法，抛出异常或终止进程。

现在一级页表中页号为0对应的二级页表无须再加载进来了，我们仅需要一级页表的两个表项和一级页表页号为1的两个二级表项，共四个页表表项，这个例子中我们所需页表表项没有改变，这是因为我们假设的页表太小了，在实际中，一旦一级页表使用未设置为0，可以有几千个二级页表项不被加载进来，极大的减小页表大小。

事实上，多级页表中每一级页表都可以设置的被恰好装进一个页，这样将不会产生任何内部碎片或外部碎片。

例如在酷睿i7中采用4级页表，每个页表9位，每一级占9位，每个页表项8字节，那么每一级页表大小是$2^{9} × 8byte = 4kb$，刚好是一个页的大小。

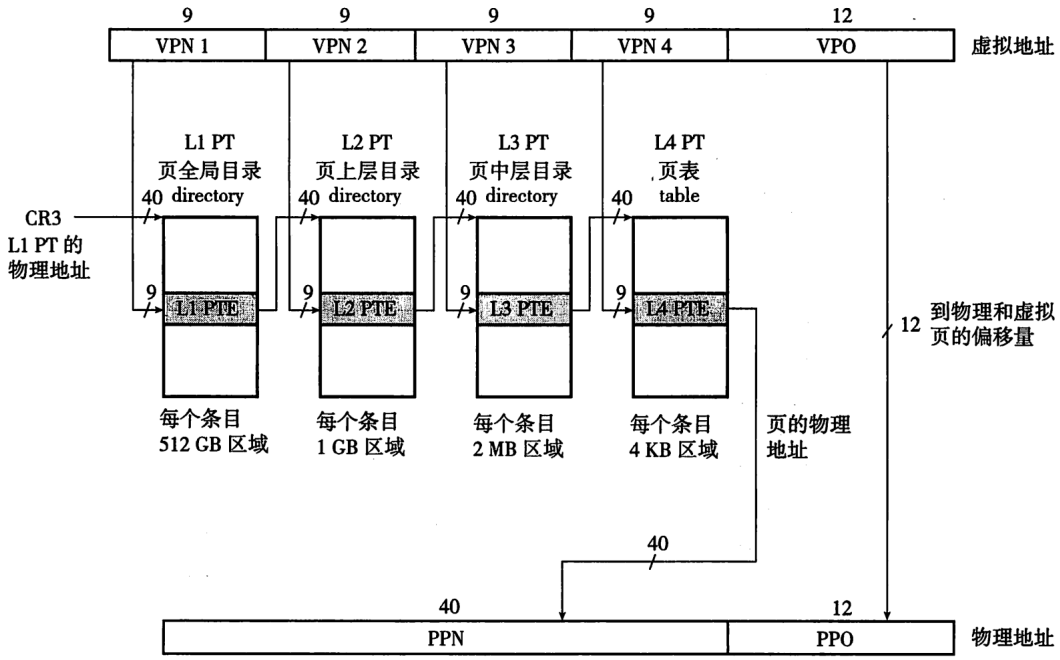


图 9-25　Core i7 页表翻译。图例：PT：页表，PTE：页表条目，VPN：虚拟页号，VPO：虚拟页偏移，PPN：物理页号，PPO：物理页偏移量。图中还给出了这四级页表的 Linux 名字

考虑一个极端的例子，如上图 VPN 为 36 位，我们极端的假设程序一点空间都不使用，那么采用上图的 4 级页表只需要装载一个大小为 4kb 的一级页表而已，而如果没有多级页表的机制，我们需要装载全部的页表项，总大小为 $2^{36} \ × \ 8byte = 512GB$！

应该指出，多级页表是有成本的。在TLB未命中时，需要从内存加载多次，才能从页表中获取正确的地址转换信息（一次用于页目录，其他用于PTE本身）。因此，多级表是一个时间—空间折中（time-space trade-off）的小例子。我们想要更小的表（并得到了），但不是没代价。尽管在常见情况下（TLB命中），性能显然是相同的，但TLB未命中时，则会因较小的表而导致较高的成本。

另一个明显的缺点是复杂性。无论是硬件还是操作系统来处理页表查找（在TLB未命中时），这样做无疑都比简单的线性页表查找更复杂。通常我们愿意增加复杂性以提高性能或降低管理费用。在多级表的情况下，为了节省宝贵的内存，我们使页表查找更加复杂。

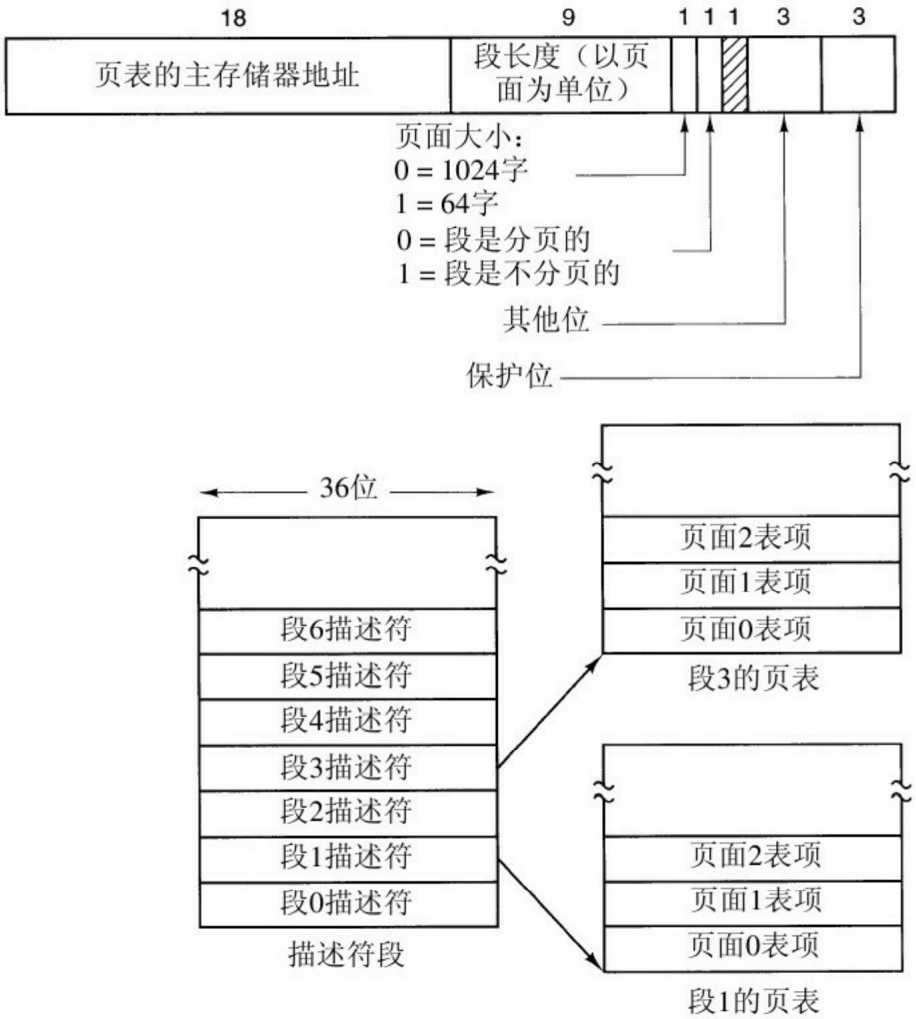
段页式存储

应该想到，在加一层抽象时，我们不仅仅可以加页，还可以加段，多年前，Multics的创造者（特别是Jack Dennis）在构建Multics虚拟内存系统时，偶然发现了这样的想法。具体来说，Dennis想到将分页和分段相结合，以减少页表的内存开销。

现在，我们仍然**将应用程序分段，但我们对于每一个段实施页式管理**，结合分段的思想，很容易可以理解为什么为什么这种想法可以减少内存开销：由于每个段都是被使用的（未使用的段不会分配空间），那么对每个段分页，至少可以保证页的使用率不会很低。

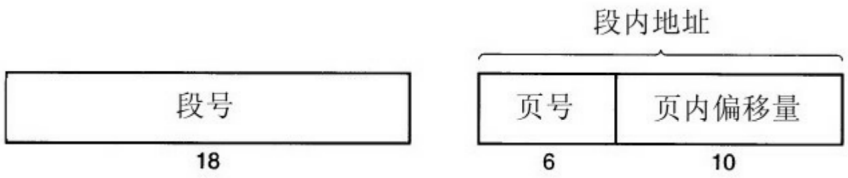
举个例子，假设程序分为代码段、堆段、栈段，4GB的虚拟空间，程序仅仅使用了15kb，其中代码段7kb，栈段4kb，堆段4kb，那么实际物理空间占用情况如图所示：

我们来思考段页式的地址转换，这需要我们结合分段与分页，此时段描述符(段表项)不再存放段基址和段长了，而是存放该段对应页表的地址，段长页存放页表的长度：



下图地址向上增长：

那么此时的虚拟地址也应该表示为 `段号 + 页号 + 偏移量` ：



我们执行如下算法：

1)根据段号找到段描述符。

2)检查该段的页表是否在内存中。如果在，则找到它的位置；如果不在，则 产生一个段错误。如果访问违反了段的保护要求就发出一个越界错误（陷阱）。

3)检查所请求虚拟页面的页表项，如果该页面不在内存中则产生一个缺页中断，如果在内存就从页表项中取出这个页面在内存中的起始地址。

4)把偏移量加到页面的起始地址上，得到要访问的字在内存中的地址。

段页式管理还使得操作系统对于某些保护或共享片段非常好管理，我们可以将一整个共享代码作为一段而不必如分页中标记页内哪些代码是共享的，此外可以发现这种管理还消除了分页管理中可能存在的少许内部碎片，同时又如分页一般，不含有任何外部碎片，易于管理，真是一个巧妙的思想。

总结

段式管理：

优点：经过巧妙的设计，可以有效的减少内部碎片，可以提高了对物理内存的利用率；将应用按逻辑分段，人们可以编写不同类型的代码，可以方便的进行共享或保护。

缺点：会产生大量的外部碎片，使得操作系统难以分配空闲空间。

页式管理：

优点：消除了外部碎片，提高了对物理内存的利用率，利于操作系统管理空闲空间。

缺点：仍然会产生内部碎片，尽管每个页碎片不超过页的大小；页表过大，占用大量空间，可以采用多级页表思想解决。

段页式管理：

优点：同时具备段式和页式的所有优点。

缺点：需要更多的硬件支持；当TLB未命中，需要更多的时间访问内存。