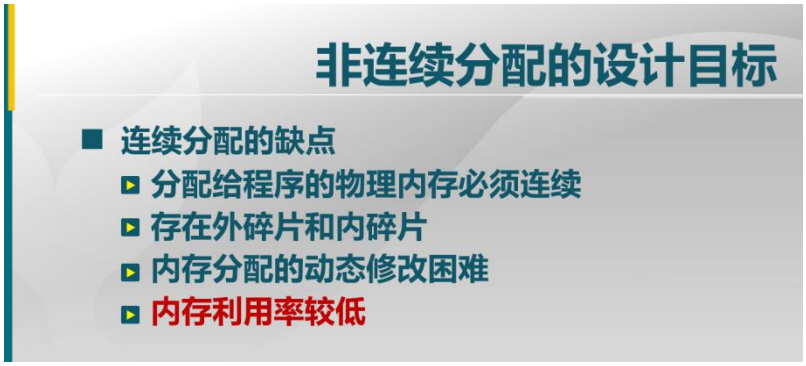
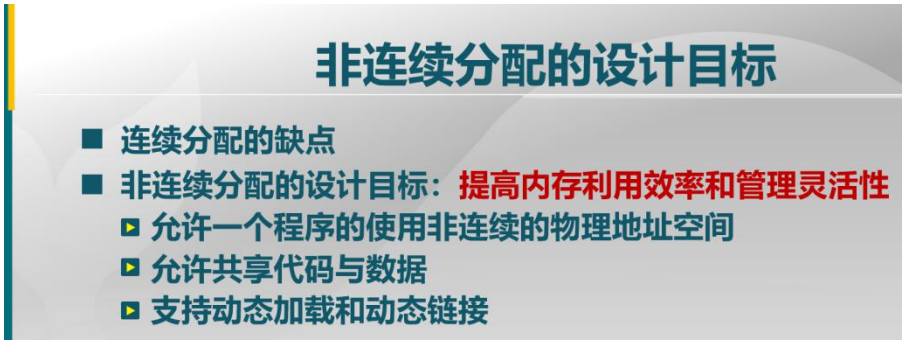
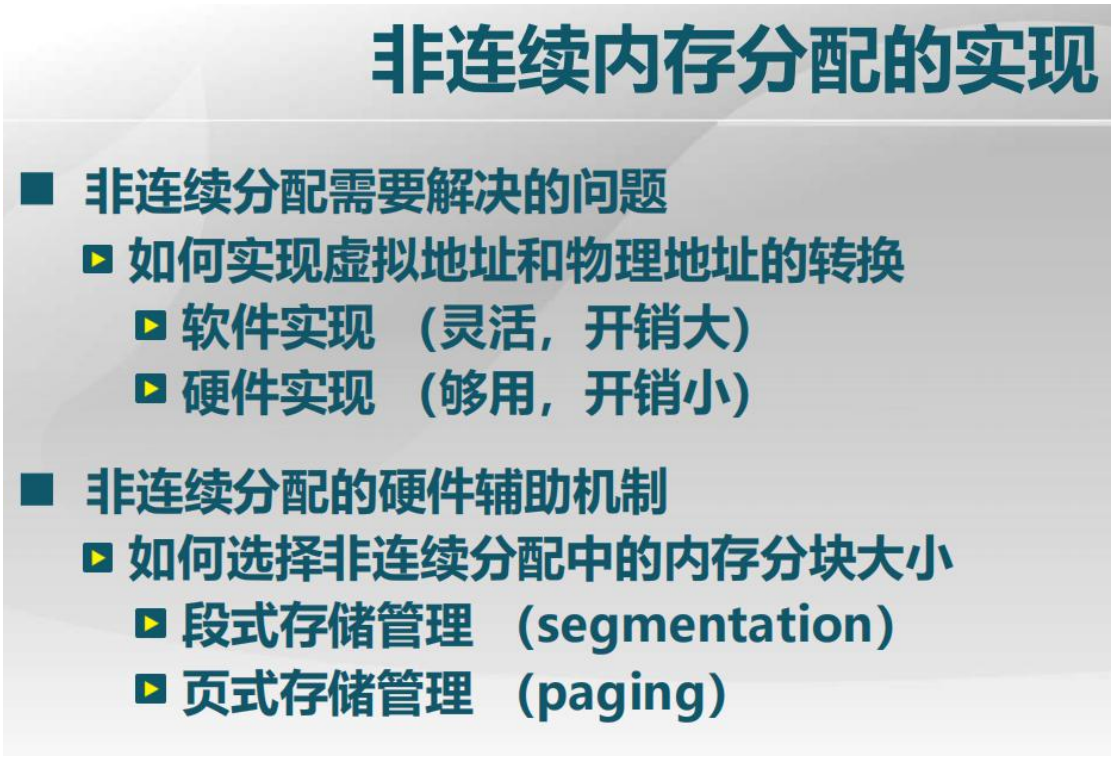
第一节 非连续内存分配的需求背景





非连续分配不是把一个字节就算成一个区域，而是受到一些限制。分配为一个字节太短了，基于这种情况，会选择不同的尺度，决定非连续内存分配每一个基本的块会有多大。基于这种基本块的大小的不同，有两种：一种是段式，一种是页式。段式分的块比较大；页式分的块比较小，分的块较小，从逻辑地址到物理地址的对应关系就会变得比较复杂，由于这个缘故这种对应关系就形成页表。还有一种方式将这两种方式结合起来，这就是我们这里的段页式。

连续分配中，分配给程序的物理内存必须连续很难达到，即使达到，在分配回收的过程中，也会产生内碎片和外碎片。如果一个程序在执行的过程中，需求的内存空间大小又有变化，连续分配很难进行动态的增加或者减少，这样最后的结果就是我们内存的利用率比较低，原因在于用户应用进程的需求没有办法满足。针对这种情况，非连续内存分配出现了，它可提高内存利用率，提高管理的灵活性。具体说来，允许程序使用非连续的物理地址空间，这样找到需要的区域的机率就会提高；在使用过程中，每个进程都要执行代码，这些进程之间有很多代码是共同的，各个进程也会用到一些数据是共用的非连续分配允许共享代码和数据以实现减少内存的使用量，例如：两个进程都要用到一个函数库，那将这个函数库的代码放到内存之后，这两个进程可以实现对函数库的共享，这样占用内存的区域就变少了；非连续分配还有灵活性的优点，可以实现某一进程分配内存大小的动态变化，实现动态加载和动态链接。

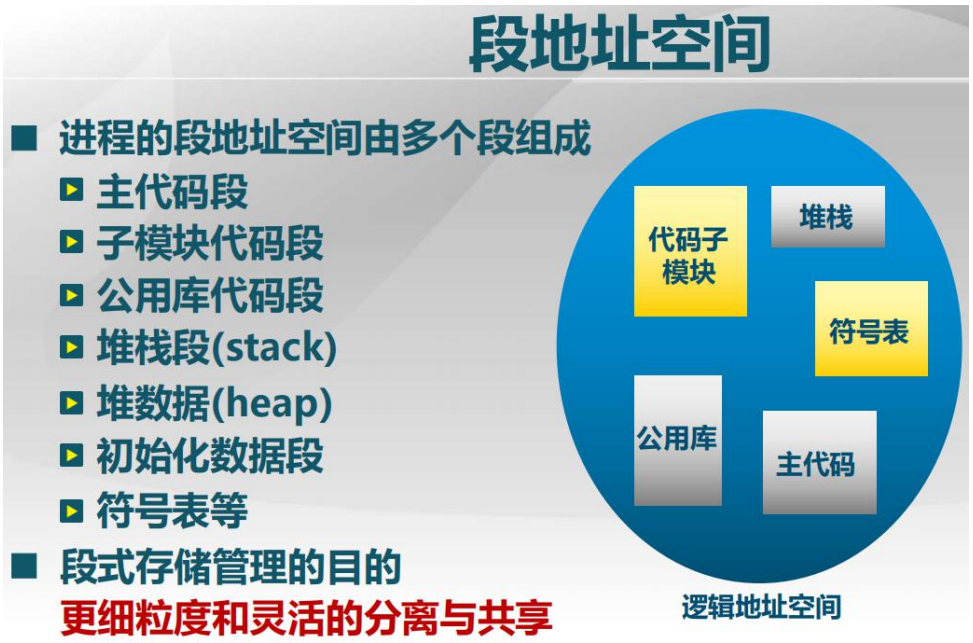


非连续分配中，虚拟地址(逻辑地址)是连续的，对应的物理地址不一定连续，这就造成了地址转换的复杂。地址转换有两种实现方式：①软件实现，比如说，要往内存中存数据，比如排序程序，由于没有办法事先确定要排序的数据的总量，这时给它分配多大存储空间都有不合适的情况，在数据结构中，我们可以先读一部分进来，排完之后放到硬盘上去，再读一部分进来，排完之后放到硬盘上去，等等，直到最后，然后把各部分排好序的重新排一遍(类似归并排序)，这就是数据结构中所说的外排序，这种方法也可以用到操作系统的内存分配中来：如果你的代码空间存不下所有代码，这时可以把其中的当前要执行代码放到内存中，把另外一部分代码放到硬盘上去，在内存和硬盘之间倒换的过程可由软件来做。②硬件实现，原因在于我们现在做的地址转换过于频繁，基本上每执行一条指令，都会去访问内存，都要做转换，这时用硬件实现，效果较好，开销比较小，而且这个转换，相对来说，它的计算过程是比较简单的重复，也就适合于用硬件来实现。

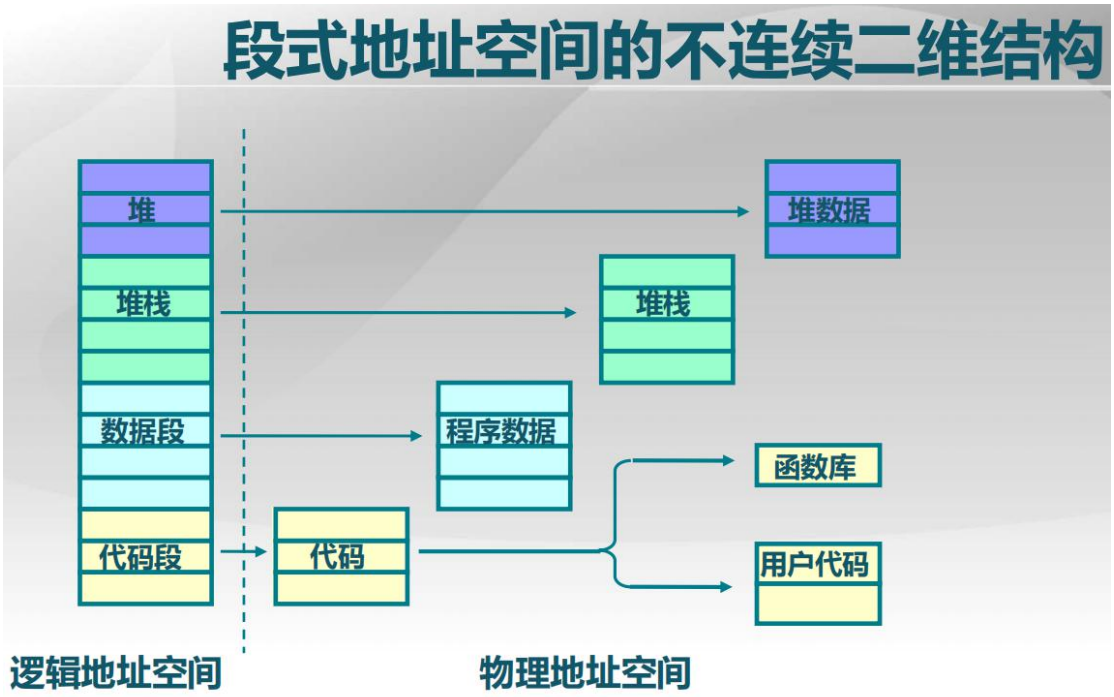
非连续分配硬件辅助机制：进程分配的内存放到不连续的地方，那每一块的大小有多大？这时有两种方法，一是段式存储，二是页式存储。简单来讲，这两者的区别就是段式存储分的块比较大，以一个段作为一个基本的单位，在分配时，一个段的内容在物理内存中必须是连续的，而不同段之间是可以放到不同地方的。页式就是分成更小的块，这个块的名字叫做页，在分配时，就是以页为单位来分配，页与页之间是不连续的。由于这两者分配的情况不同，实际上两者在实现时会有很大的区别。

第二讲 段式存储管理

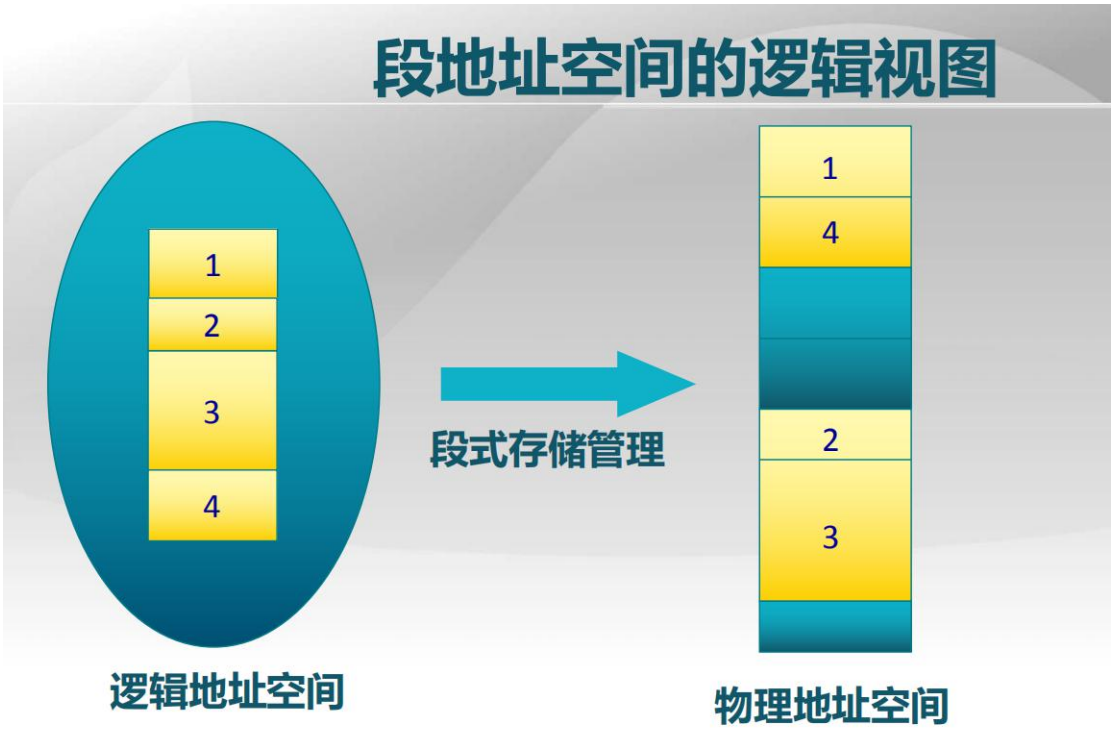
本节将说明段的存储空间是如何来组织的以及在段式存储管理当中，内存访问是如何进行的。



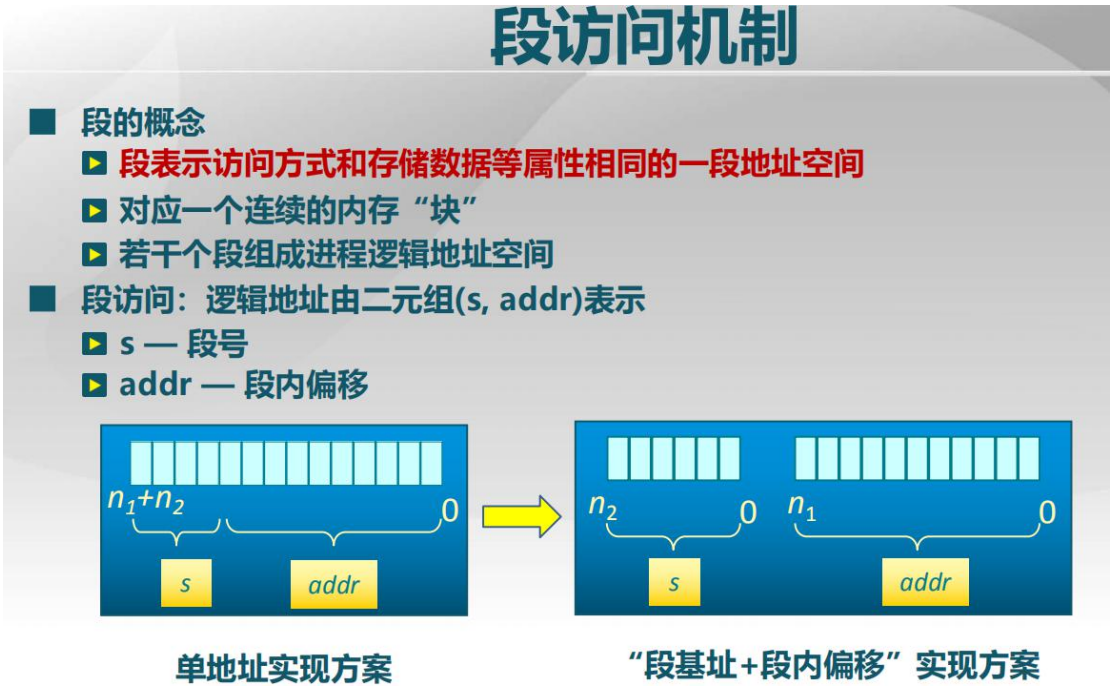
在段式存储管理当中，我们把进程的地址空间看成由若干个段组成的。每一段在上图中有一些实例，比如说我的程序有主程序、有子模块，各个子模块可以看成是相对独立的一个段，主代码看成是一个段，公共库看成是另外一个段，以上是代码。同时还有一些堆，初始化的数据，堆栈，符号表，这样一些数据段，在这里，也可将其分别视为一段。



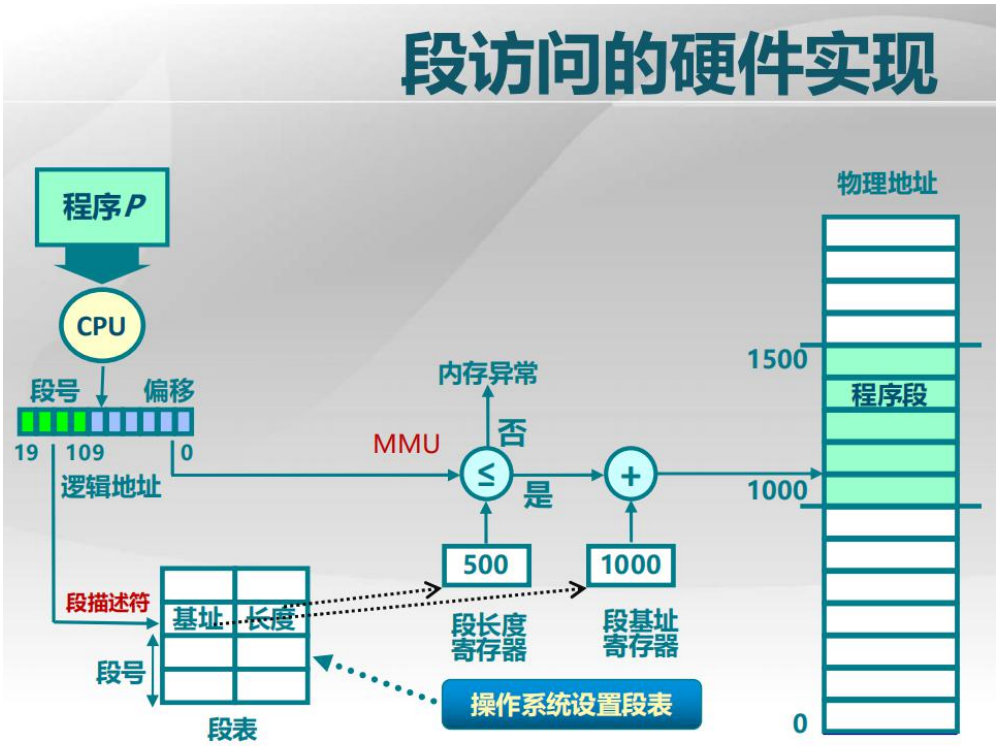
上图，利用段式存储管理，将逻辑地址空间转换为一个不连续的二维结构。



上图展现了段地址空间的逻辑视图。



每一段对应一个连续的内存块。



上图解释：上图最右侧是进程的物理地址空间，左侧，程序在 CPU 上执行，它要访问一个存储单元的时候，首先在段地址空间中的逻辑地址分为了两段：段号和段内偏移，首先用段号去查进程的段表，若干个段，所有段都在段表里有一项，每一项对应一个段描述符，这里的基本内容是段的起始地址(基址)和它的长度。段表里的内容可以用操作系统的软件来对它进行控制。此时硬件部分，存储管理单元 MMU 把段表中的长度和偏移取出做比较，查看偏移是否越界，如果越界就会出异常，否则，是一个合法的访问，在 MMU 中将段基址和偏移加到一起就可找到实际要访问的物理地址了。以上就是基于段机制进行访问和工作的过程。

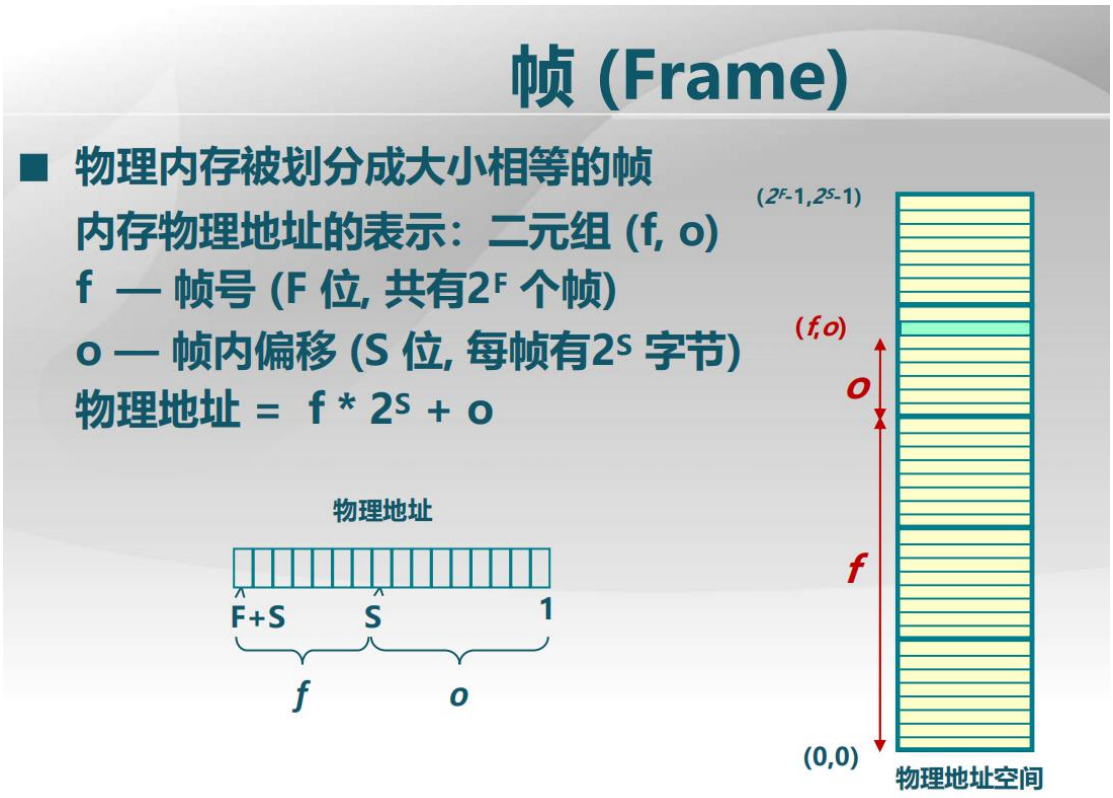
第三节 页式存储管理

本节介绍页式存储管理的概念以及页式存储管理的地址转换(映射)机制。

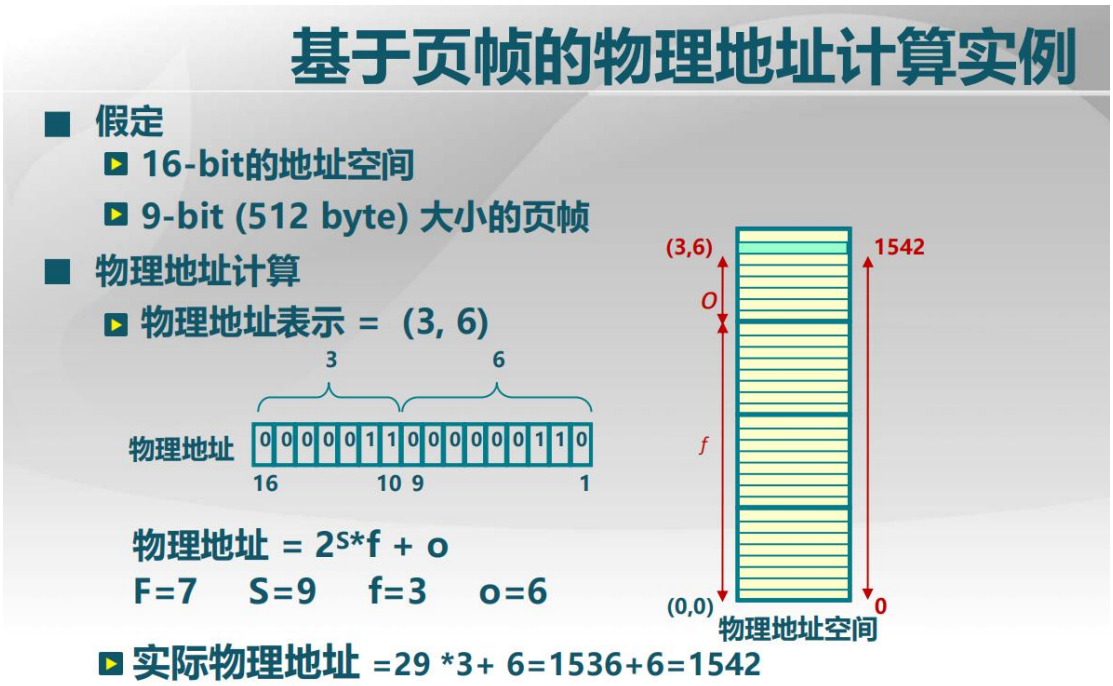
在页式存储管理当中，它把物理的地址空间分成的基本单位叫做页帧或者叫帧(frame)，页帧的大小为 2 的 n 次方，原因在于，我们要在地址转换的过程中，让转换的过程比较方便，

在计算机中，二进制的移位是使得乘法非常快速的一个因素，比如在 32 位机器中，4K=4096B 是我们常见的一种页帧的大小。　　与此同时，逻辑地址空间也要分成大小相同的基本单位，这就是页(Page)，frame 和 page 的区别在于一个用来描述物理页帧，一个用来描述逻辑页面。

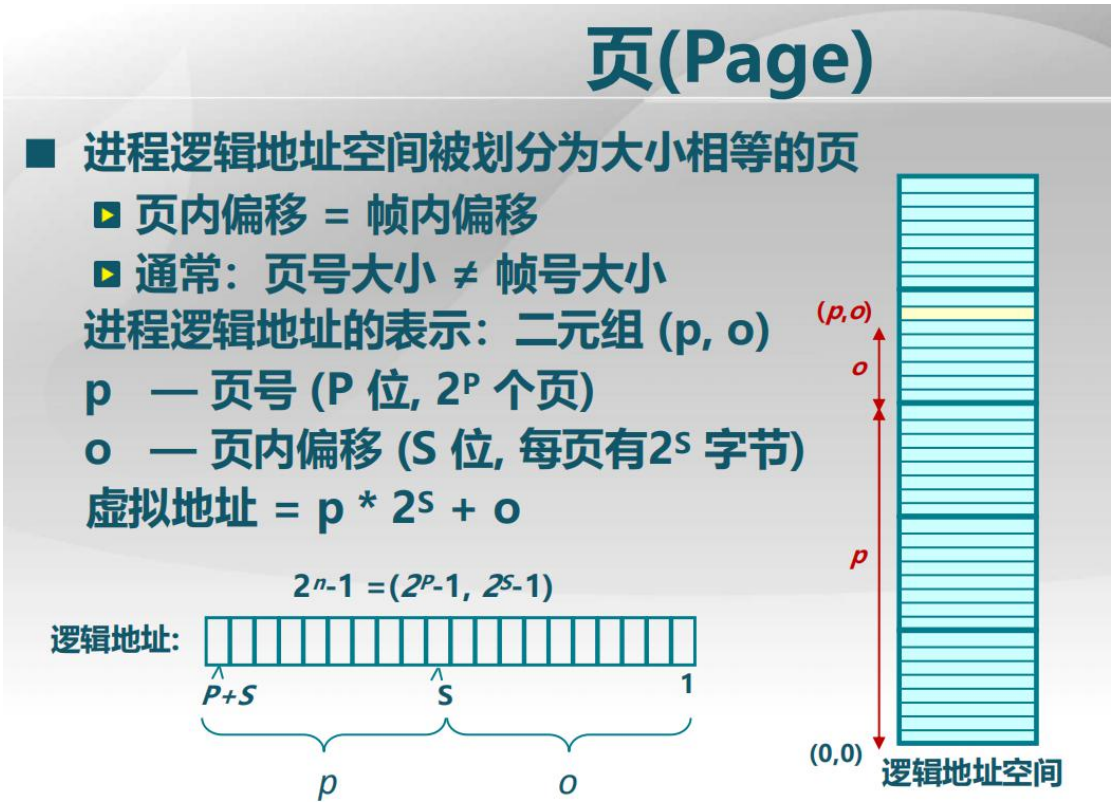
页面(逻辑地址)到页帧(物理地址)的转换如何进行？这个转换涉及到页表，其用于保存转换关系。有了这个转换关系后，如何让这个转换能够高效地进行呢？这就涉及到 MMU(存储管理单元)和 TLB(快表)



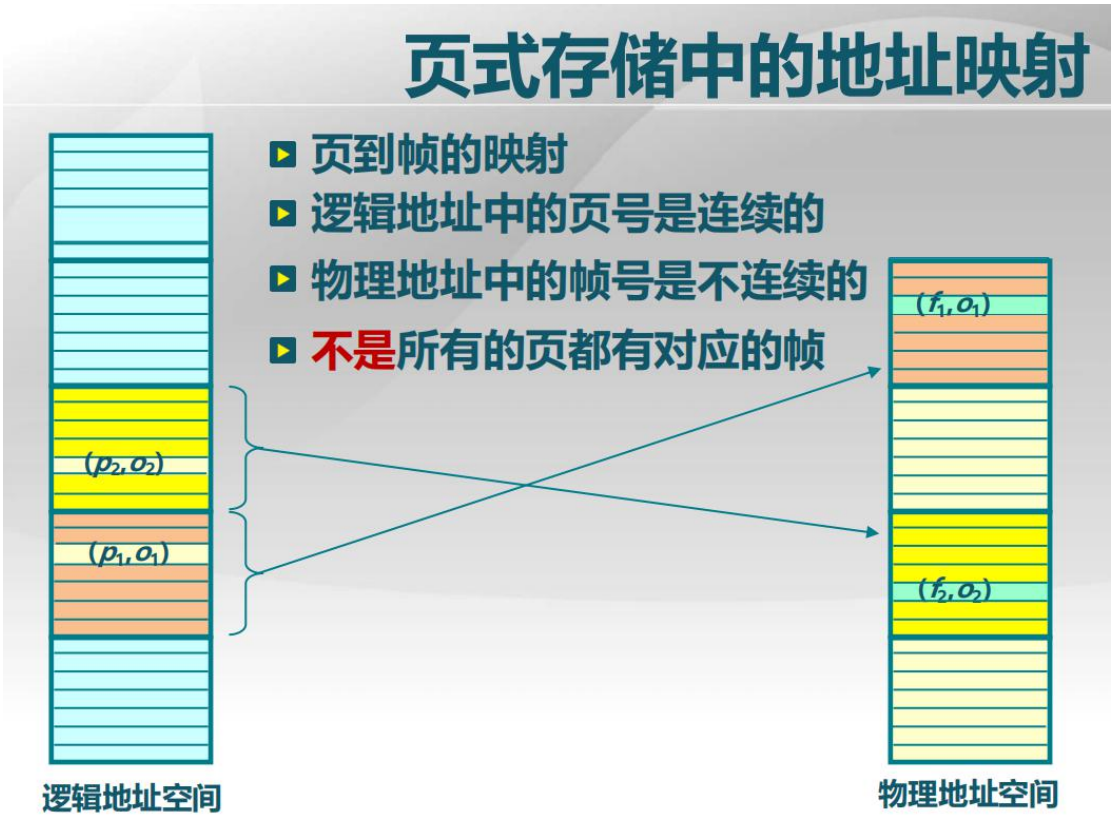
页帧是物理内存基本块的名称。上图说明了如何将一二元组转换为物理地址，下图是一个计算的实例：

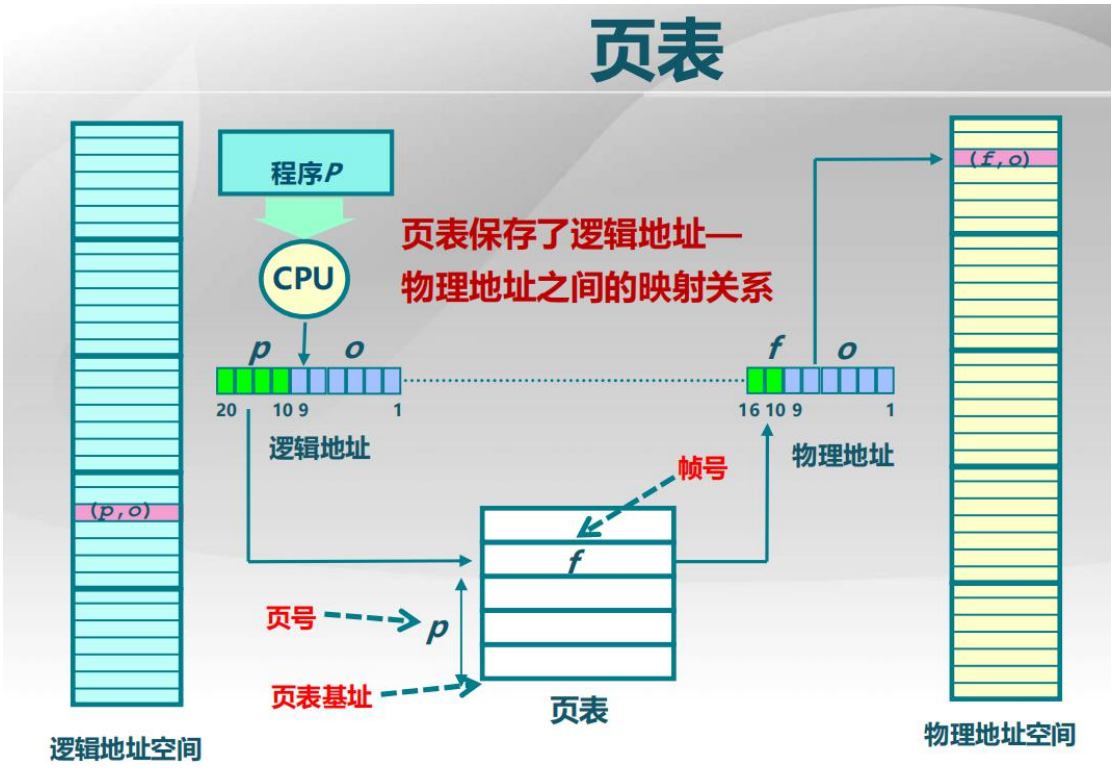


可见偏移 o=6,帧为第三帧，则实际物理地址为 512*3+6=1542(PPT 上写的 29，写错了)



上图显示了逻辑地址空间的划分方法，其余物理空间的划分是类似的，由于一个页帧和一个页面的大小是相同的，所以页内偏移=帧内偏移；由于逻辑地址空间的页号(x)是连续的，对应到帧号(y)不一定是简单的 x=y 映射过去，所以页号和帧号通常不相同。

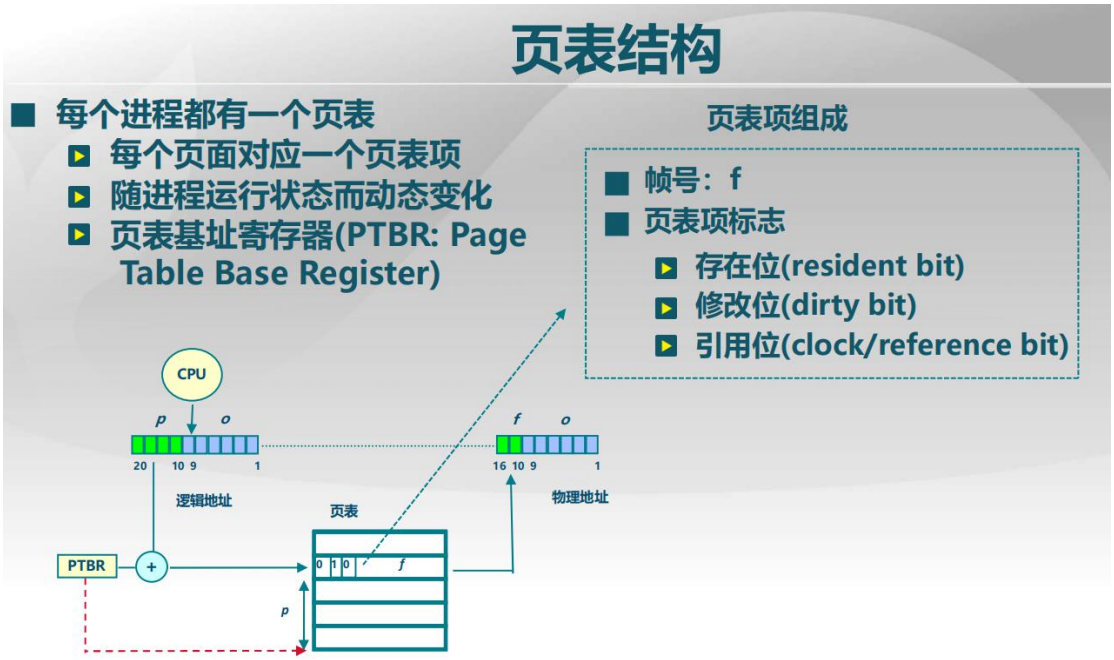




令逻辑地址空间中的页号是 p，物理地址空间中的帧号是 f。程序在 CPU 中执行，执行时得到的地址是(p,o)，即逻辑页号和页内偏移，用 p 到页表中去找相应的 f，页表保存了逻辑页号到物理页号的对应关系，这个表由页标基址来指定它(页表)的开始位置，用页号作为下标去查找这个数组就能找到相应的页表项，每一个页表项有一个固定的长度，帧号是页表里存的字段之一。(f,o)是实际的物理地址。(o 占 s 位)

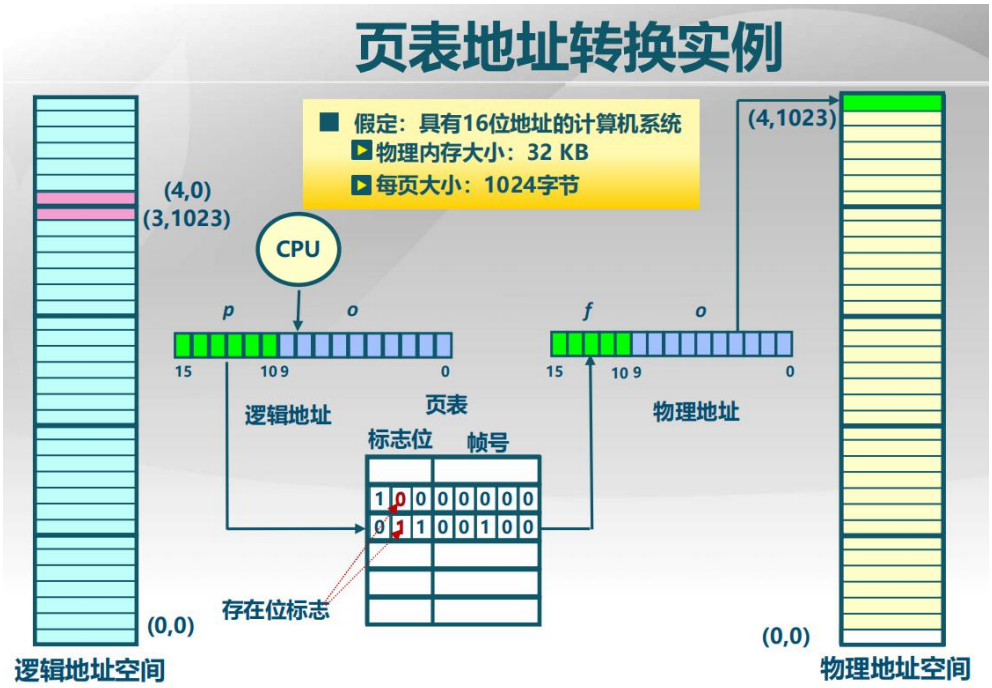
第四讲 页表概述

页表是负责从逻辑页号到物理页号之间的转换。这个转换究竟是如何进行的？

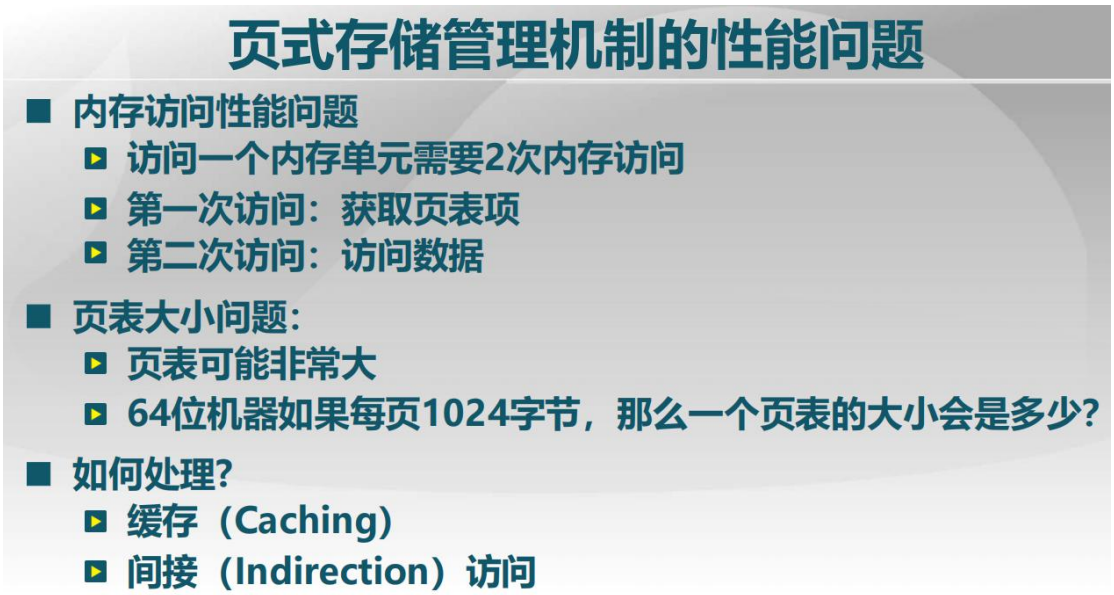


每一个逻辑页面在页表中对应一个页表项，这个列表项完成逻辑页号到物理帧号的转换。在这个转换过程中，页表里的内容会随着程序的运行而发生变化，这种变化使得我们有可能动态的去调整分配给一个进程的内存空间的大小。页表存放在一个寄存器里，即页表基址寄存器，这个寄存器将告知页表的起始位置在哪里，有了逻辑页号之后，就可以找到相应的页表项。一个页表项中有页帧号(帧号)，存在位：是指一个逻辑页号是否有一个物理帧(物理页面)和它相对应，如果有，这个存在位就是 1.修改位：对应的这个页面里面的内容是否修改了。引用位：这个页面在过去一段时间里是否有对它的引用，即是否访问过这个页面里的某一个存储单元。

以上，我们在页表中加了几个标志位，下面，通过一个实例来看这几个标志位当中的存在位在页表及页式存储中的作用。



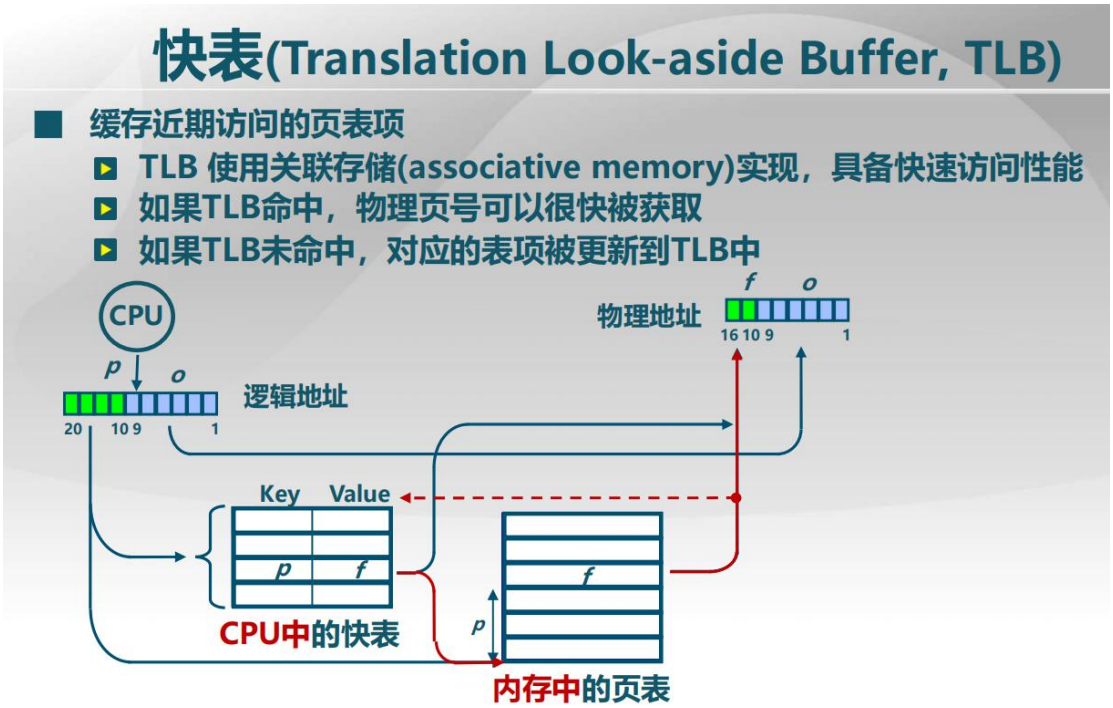
假定即每 1K(1024 字节)算一页，逻辑地址，0-9 是页内偏移，10-15 是页号。页表在此处完成逻辑地址到物理地址的转换。在没有标志位的情况下，是每一个逻辑页号都对应有一个帧号，现在有了存在位后，有的页可能没有对应的物理帧号(标志位对应为 0)，这时就相当于没有给进程分配相应的存储，这使得我们在这里可以有动态的变化。



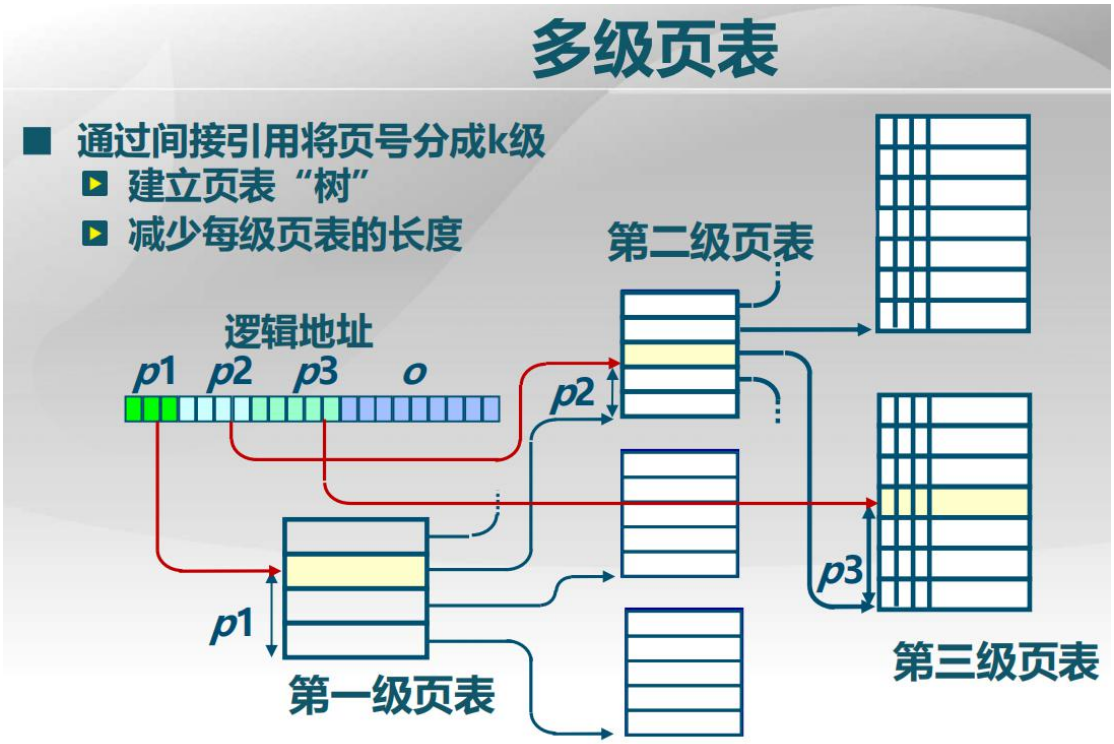
页式存储管理可以让我们不连续地分配内存空间，但它也会带来很多的问题。如上图所示。内存访问性能问题造成读写性能大幅度下降，读写量大幅度增加。64 位机器如果每页 1024 字节，这时如果仍然使用 1K(1024 字节)作为页面的大小，那将有 2^(64)/2^(10)=2^(54)个页表项，即 2^(54)个页面。　　处理方法：缓存，由于程序执行时，访问的数据和代码都具有一定的相邻性，这样把得到的页表项缓存下来，下一次访问时利用这个缓存，有极大的可能性可以直接访问到物理内存，这样可以把访问页表的次数减下来。间接访问：页表很大会很麻烦，将页表切段间接访问，即先找在哪个子表里，再对子表进行查找等等，这种间接访问对应过来就是多级页表。

第五讲 快表和多级页表

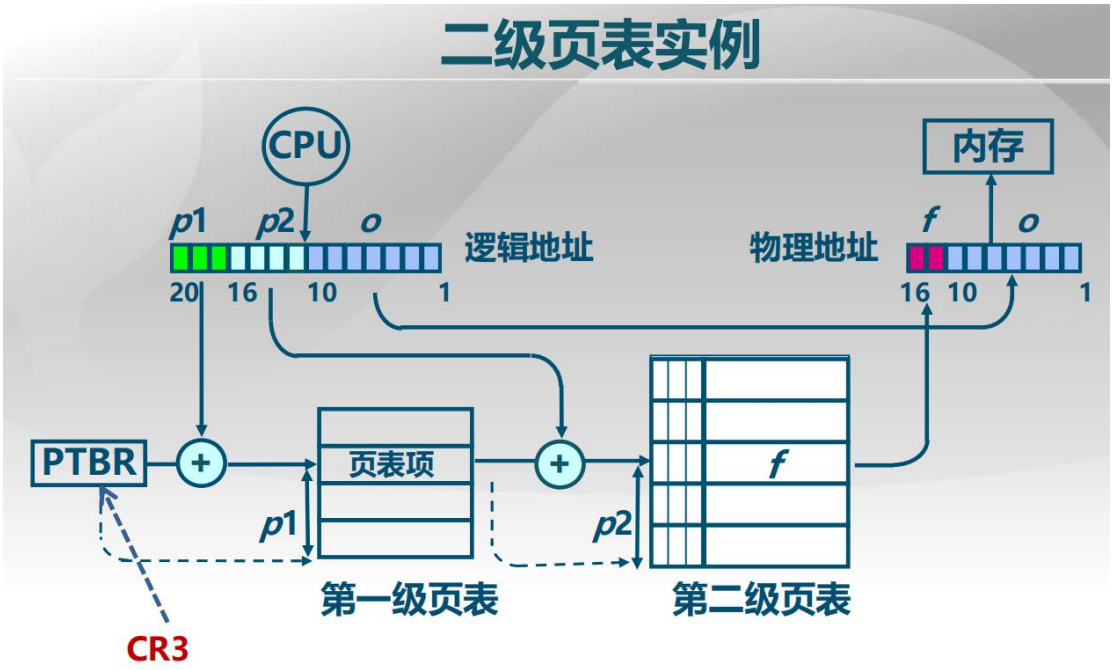
快表是利用缓存的机制来减少对内存的访问，而多级页表是通过间接引用的方式来减少页表的长度。



快表，实际上就是把近期访问过的页表项缓存到 CPU 中。CPU 中的快表查询比内存中页表的查询快得多，这是由于 CPU 的性能比内存高很多，相应地，CPU 中的快表不能做很大，因为 CPU 成本高，功耗大。快表中只是保存近期访问的页表项。关联存储可以实现并行(并发的关键是你有处理多个任务的能力，不一定要同时。并行的关键是你有同时处理多个任务的能力)的同时查所有的快表项。引入快表后，先在快表中查，查不到再去页表中查，同时更新相应项到快表中，而不像以前直接查页表了。快表中逻辑页号是 key。



多级页表是通过间接引用，将页号分成若干级。比如上图中，原来的逻辑地址的格式是页号(p)加页内偏移(o)，现在变成了三级页号(p1 p2 p3)加页内偏移(o)，和这种变化相对应，我们的页表会因此而形成一个树状结构，即原来一张大的线性页表，将它切成若干段。如果所有的页表项都存在，多级页表并没有减少存储，但实际上运行的进程，并不是所有页表项都有必要存在，这时可以通过各级页表当中的存在位，把那些不存在的页表项省略掉。例如在第一级页表中有存在位表明下一块区域都不存在，这样此一级页表项就没有对应的二级页表，这样节省出来的空间就大幅度增加。因此，使用多级页表可以有效减少页表的大小。

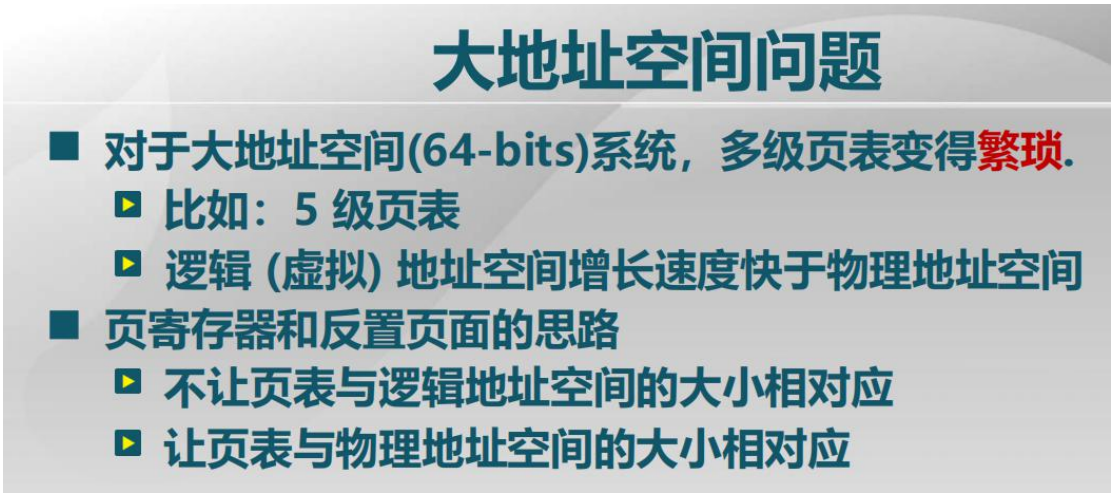


上图是一个二级页表的实例。这里将 20 位地址总线切成了 3 段。在因特尔 CPU 上，有

一个 CR3 寄存器，存有第一级页表的起始位置(ptbr)，然后加上第一级的页号(p1)，作为查找的索引(下标)，便找到第一级页表中相应的页表项，页表项中有对应的第二级页表的起始位置。此时第二级页表的起始位置加上第二级页表号(p2)，得到第二级页表中相应的页表项，页表项中有对应的实际的物理帧号，这时再把偏移搬过来，就得到物理地址了。

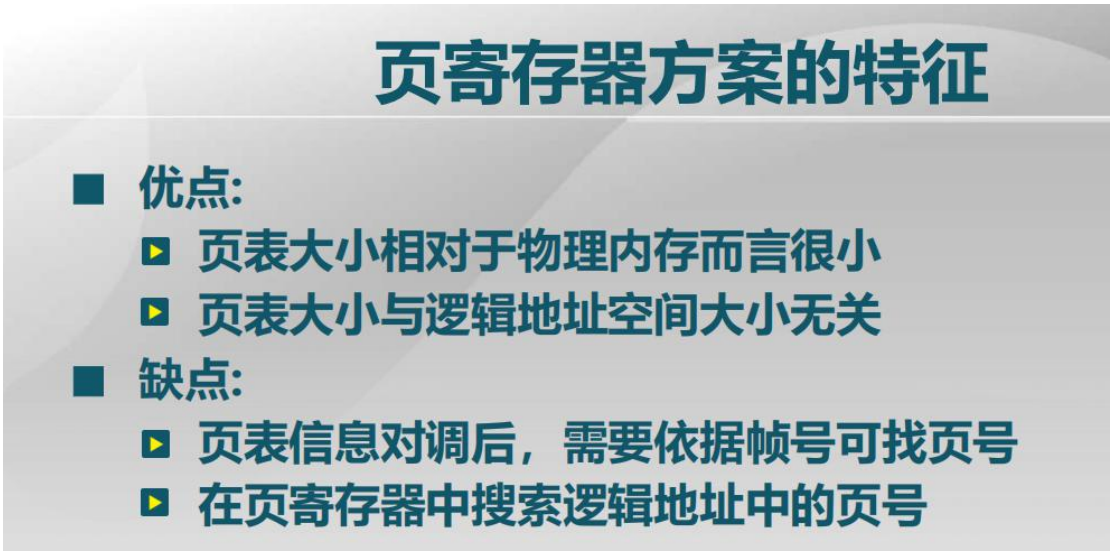
第六讲 反置页表

反置页表也是一种为了减少页表所占用存储空间而产生的一种做法。

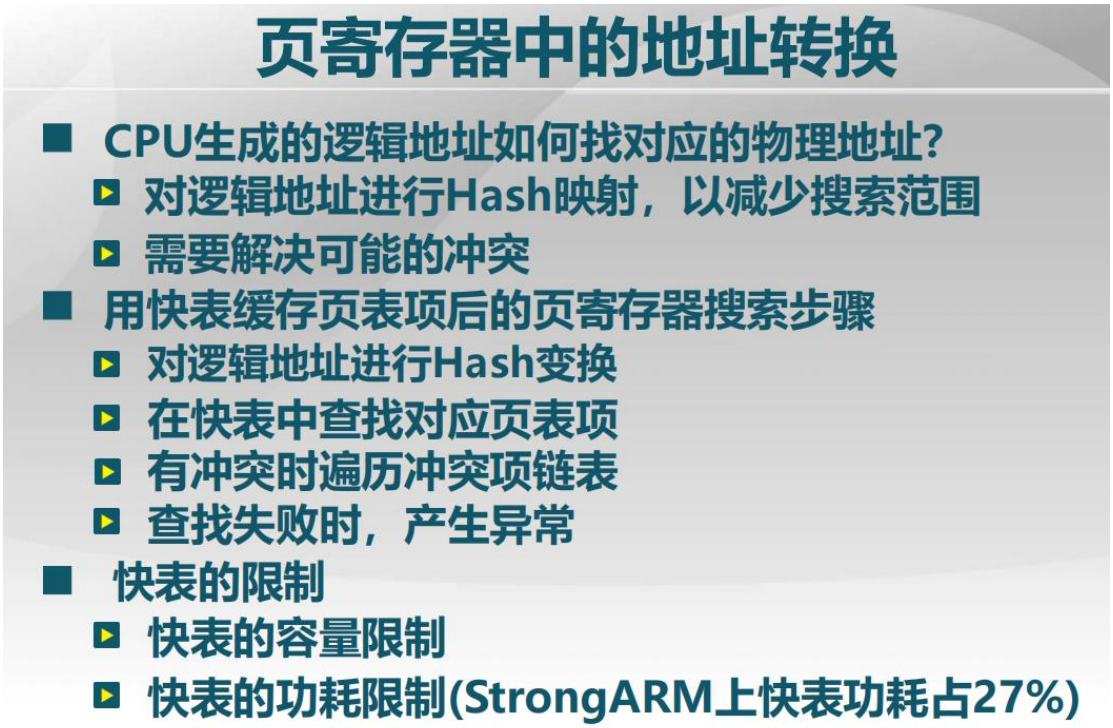


对于大地址空间，多级页表访问存储空间的次数会比较多。　针对多级页表的问题，反置页表(面)和页寄存器是解决其问题的两个类似的做法。它们的做法是让页表项和物理地址空间的大小相对应起来(多级页表是和逻辑地址空间相对应起来，这样因为逻辑地址空间增长很快，页表就会很大)，这样的话，随着进程数目的增加和虚拟地址空间的增大都对页表占用的空间没有影响。

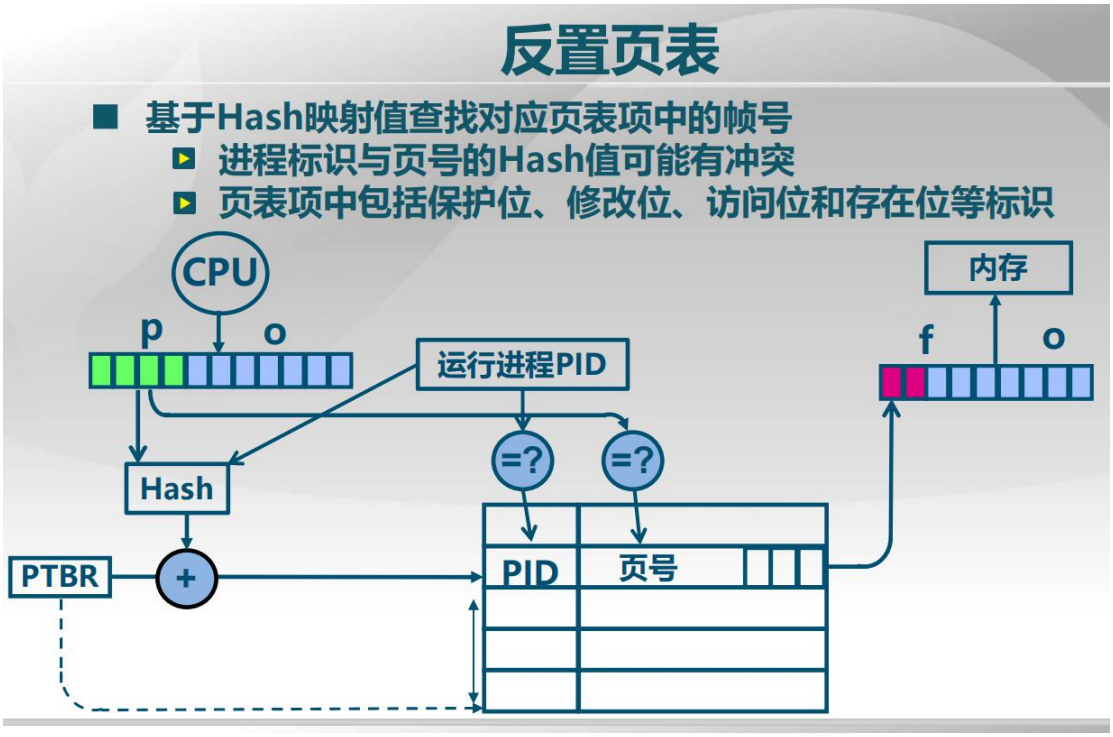
上图展示了页寄存器的实现。每一个物理帧和一个页寄存器相关联。占用页号即一个进程占用了这一页，该物理帧对应的逻辑页号是多少，这样就可以知道这个物理帧分配给了哪一个进程以及逻辑地址是多少。保护位用来约定这一页的访问方式，比如可读、可写。



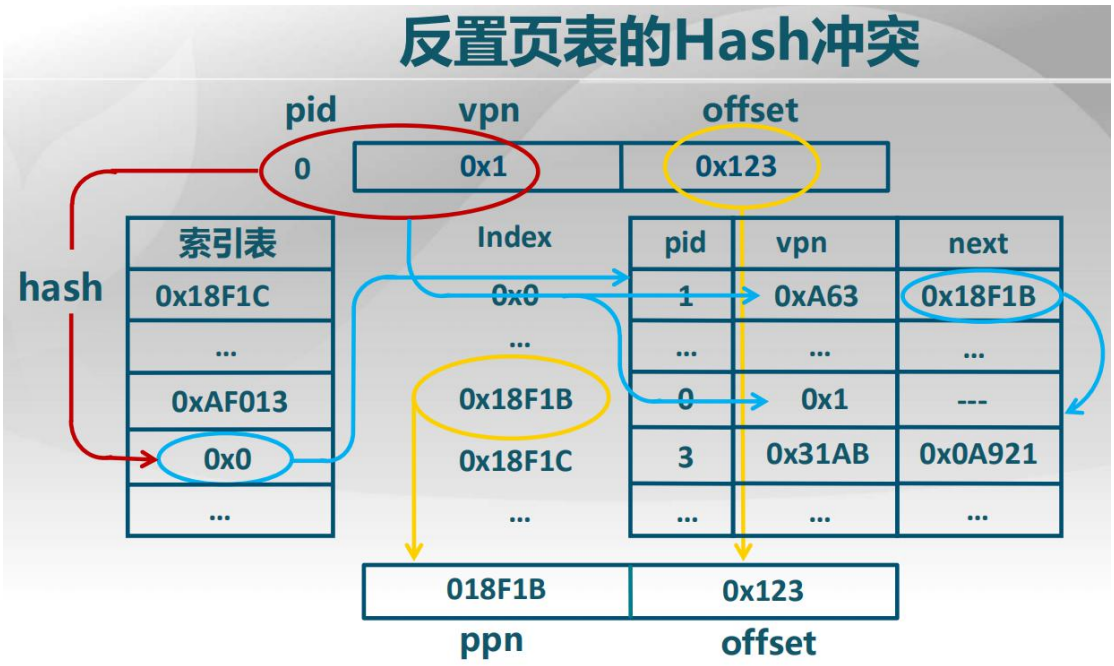
缺点：在多级页表中，信息查询是从逻辑页号到物理页号，根据下标查找数组即可，不需搜索(从已知的进程对应到物理内存)，现在反其道而行，为了找到对应的进程，需要对页寄存器进行搜索，开销很大。



hash 后可能两个不同的逻辑地址对应到同一个 hash 值上，因此需要解决可能的冲突。这时可将快表和页寄存器机制一同使用。



上图介绍了另一种方法：反置页表。反置页表和页寄存器的区别是：反置页表把进程 ID 即 PID 也考虑进来了。反置页表将进程标识 PID 和逻辑页号 p 一块 hash，hash 完的结果也可能有冲突，需要解决冲突。hash 后的结果是以页帧号来排序的，需要和反置页表项核对，hash 完之前的结果和之后的里面的进程 ID(PID)和逻辑页号是否一致，如果一致，那这就是要找的那一项，便可以得到相应的物理帧号了，如果不一样，就会产生冲突，对于冲突情况，我们以下图为例：



进程标识(pid)和逻辑页号(vpn)做 hash，hash 完的结果是索引表中 0x0，到相应的反置页表当中进行查找 index(此索引即为页帧号，因为 hash 后反置页表里的结果是以页帧号来排序的)为 0x0 的项，看进程 id 和逻辑页号，此时 pid=1 不等于 0，vpn=0xA63 不等于 0x1，说

明产生了冲突，这时按照 next 中的指示，查找索引为 0x18F1B 的项，可见此时 pid 和 vpn 都能对应上了，因此 0x18F1B 就是所需的页帧号。此时页帧号和页内偏移合到一起就得到物理地址了。

以上，介绍了三种来缓解页表带来的麻烦：

①快表，其通过缓存的机制，来减少对物理内存、对页表的访问

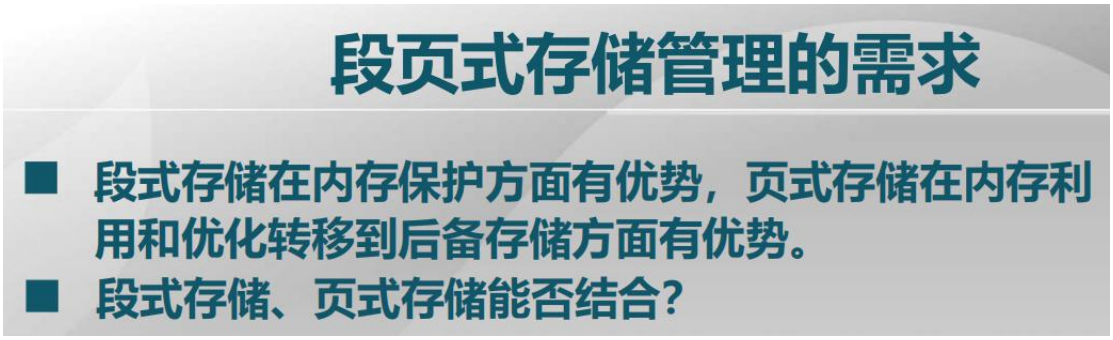
②多级页表，通过多级来减小页表的大小

③反置页表是另一种减小页表大小的做法，因为反置页表中只有物理地址那么多的项，而不是寻求逻辑地址和物理地址的一一对应关系。

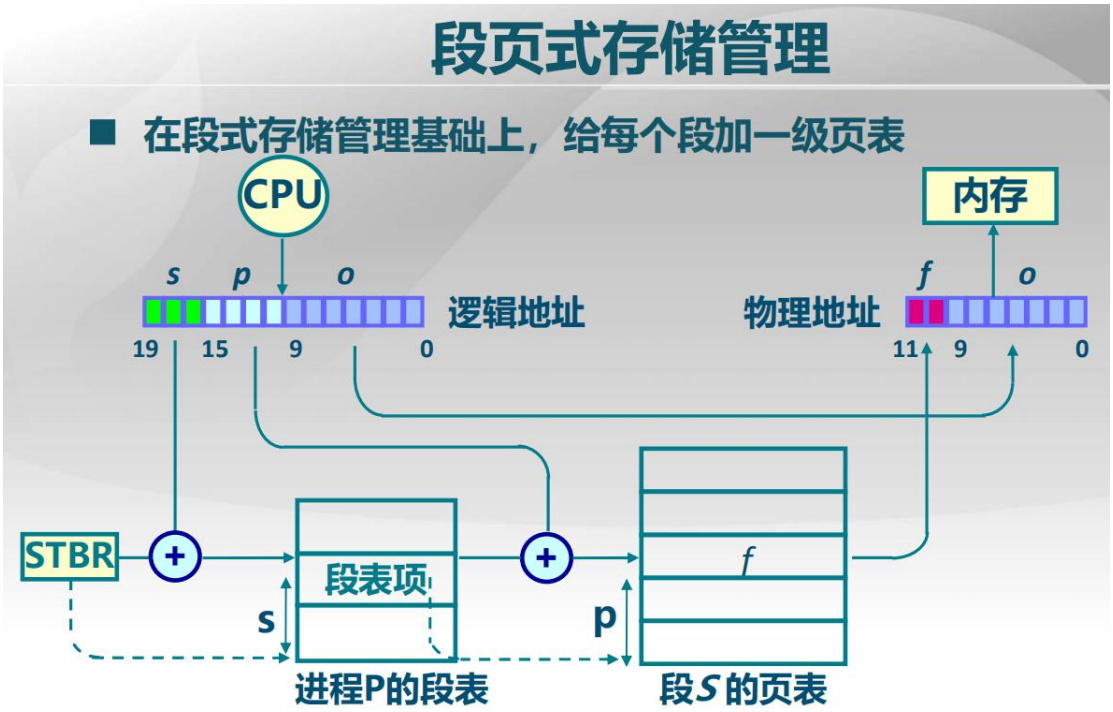
这几种做法可缓解我们引入(普通)页表所带来的麻烦，我们现在现在用到的计算机，都是用以上机制进行转换的。以上，就是页存储机制的讲解。

第七讲 段页式存储管理

段页式存储管理是我们前面段式和页式做结合的结果。

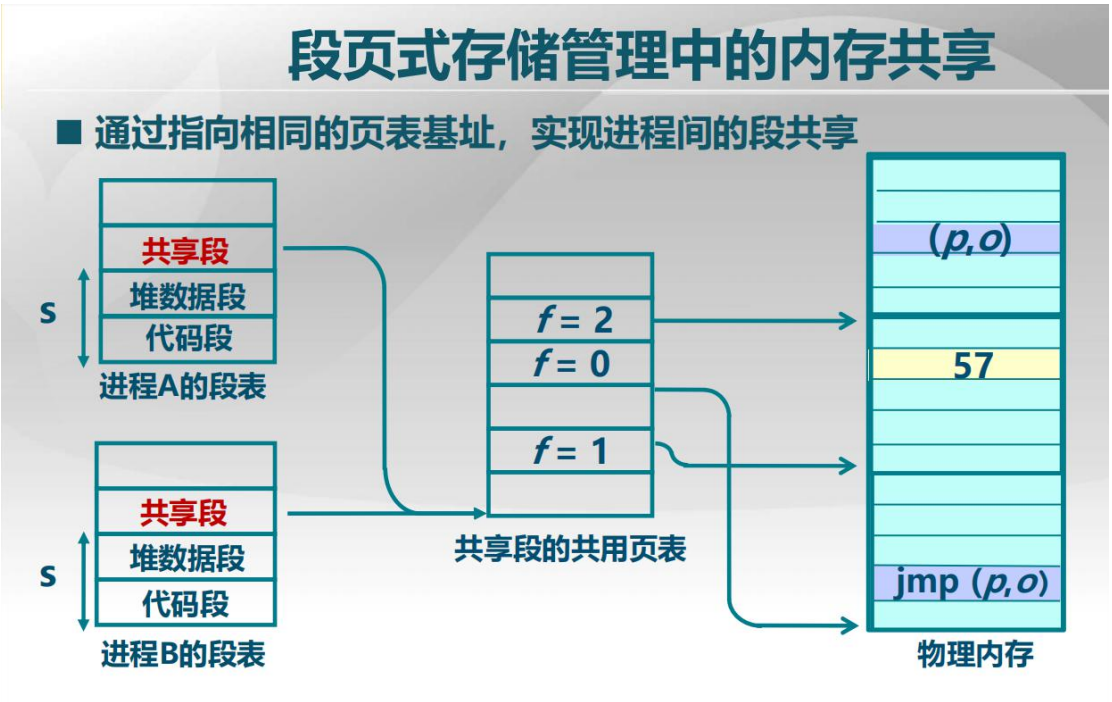


由于段式存储分的块比较大，每一块里存储的内容是同一个段，这同一个段的访问方式和存储的数据都是相同的或者类似的，这样，去做存储的保护比较方便。页式存储分了很小的标准大小的块，它的内存利用效率及内存转移到外存方面有优势。



段页式中，逻辑地址变成段号(s)、页号(p)加页内偏移(o)，如果在页表中是多级页表，p

还可分成 p1 p2 等等多级，从逻辑地址，最后变到的物理地址是物理帧号(p)加页内偏移(o)。具体做法：由进程的段基址找到相应的段表基址，加上段号，找到相应的段表项，段表项中，有相应段的段长度和段基址，这些加在一起，得到相应段的页表的基址，页表的基址加上页号得到相应的页表项，页表项中就有对应的物理页帧号，页帧号和页内偏移加在一起就可以访问到实际的物理存储单元了。这种做法可以在其基础之上很方便的实现内存共享：



即在段表的基础上，加上一个共享段，使得两个进程可以指向同一个页表，那么这两个段就共享了。上图是一个段页式的访问方式。在页式、段式、段页式里共享的方式都是类似的。

以上，我们介绍了非连续内存分配的几种做法：段式，页式和段页式，它们的共同点是：分配给一个进程的内存块可以是不连续的，区别在于：各自分配块的大小有不同，段式分配的段是很大的，以一个段为单位；页式分配的块很小，以 1K，几 K 这种尺度作为一个最小单位，而段页式是把这两种结合起来，在这种非连续内存分配做法里，面临的问题就是中间要加一级页表或者段表，这个表的加入会使得原来的连续存储方式所没有的一些问题在这里出现，比如访存过多，页表的大小很大，针对这些问题，我们又有一些列做法：如快表做缓存，多级页表，反置页表做间接存储访问，这些做法都可以改进由于引入非连续存储所带来的麻烦。

以上所讲都是一些基础做法，在实际系统里，基本原理是类似的，但其中很多实现细节需进一步了解。

习题：

1.可有效应对大地址空间可采用的页表手段是()

A.多级页表

B.反置页表

C.页寄存器方案

D.单级页表

前两个是对的。至于为什么页寄存器不行：

页寄存器和反置页表很像，但它们的一个区别是进程 ID 在地址转换中的使用。没有进程 ID（也就是说页寄存器方案）时，页表占用的空间仍然是与进程数相关的（也就是每个进程对应一组页寄存器？）。反置页表的大小只与物理内存大小，与并发进程数无关。

采用页寄存器的硬件开销会很大。所以现在的通用 CPU（包括 64 位的 CPU）没有采用这种方式，大部分还是多级页表。由于有 TLB 作为缓存，效率还不错。

2.描述段管理机制正确的是()
A.段的大小可以不一致
B.段可以有重叠
C.段可以有特权级
D.段与段之间是可以不连续的

都对。段的大小显然可以不一致（段描述符中给出的大小不同）。段之间可以重叠（没说不能重叠，而且完全扁平模型就是全都映射到全部物理内存。结合段的实现机制中，只判断偏移是否大于段长度理解）段可以有特权级（段描述符中的 DPL，访问段的最低特权级）。段之间当然也是可以不连续的。

3.描述页管理机制正确的是()
A.页表在内存中
B.页可以是只读的
C.页可以有特权级
D.上述说法都不对

前三个都对。当然有的地方不太准确。在 80386 系统中，一级页表一定在内存中，但二级页表不一定在内存中。PDE 和 PTE 都可以规定访问权限，不过只有 U/S（用户/OS 权限）和 R/W（只读/可读可写）位.

4.页表项标志位包括()
A.存在位(resident bit)
B.修改位(dirty bit)
C.引用位(clock/reference bit)
D.只读位(read only OR read/write bit)

都有。当然，还不止这些。

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

NOTES:

1. See Section 4.1.4 for how to determine whether the PAT is supported.