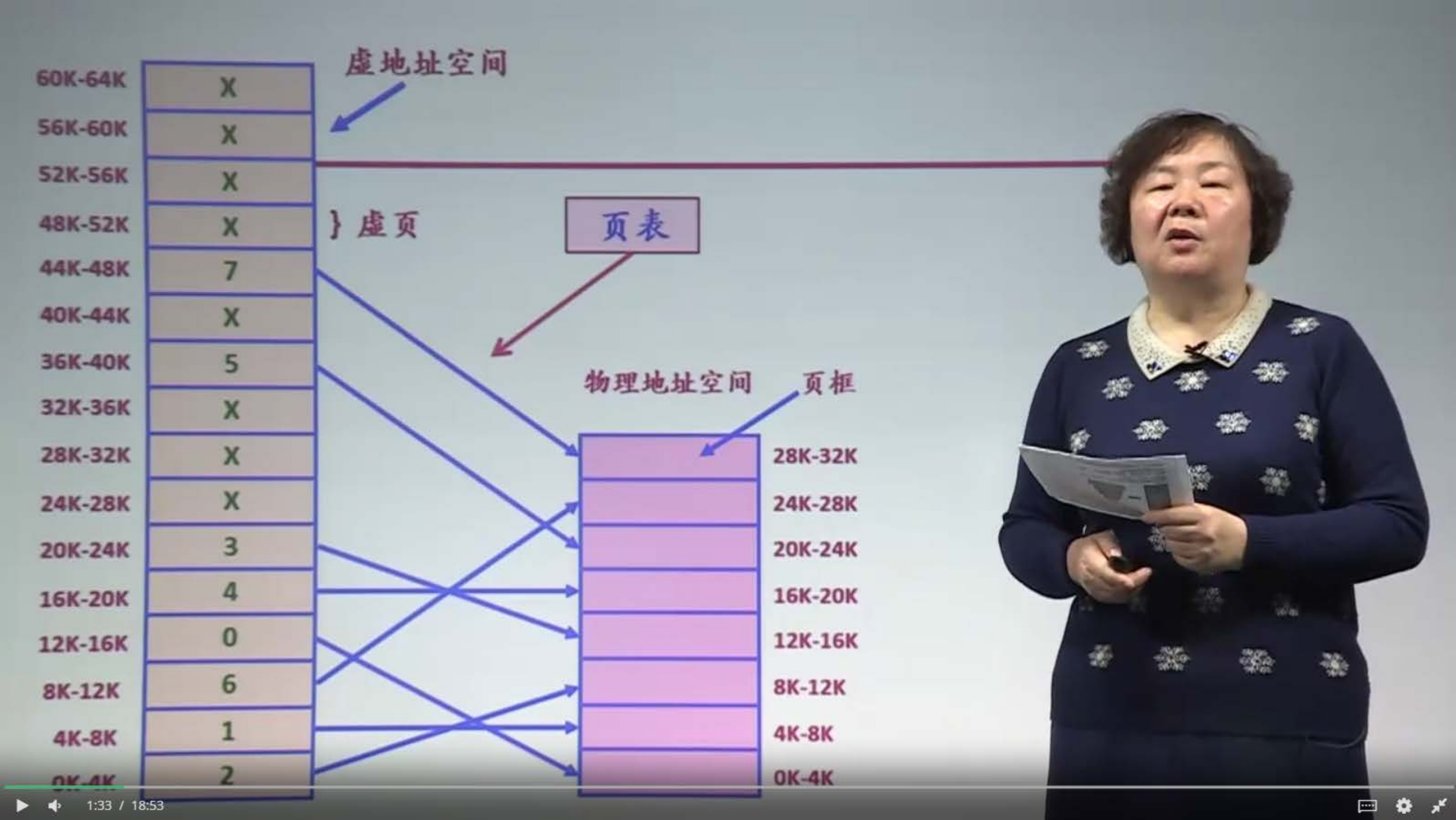
多级页表、X86的页表项示例

页表及页表项的设计

下面我们介绍，在虚拟页式存储管理方案当中 页表及页表项的设计问题


虚地址空间
60K-64K
56K-60K
52K-56K
48K-52K
44K-48K
40K-44K
36K-40K
32K-36K
28K-32K
24K-28K
20K-24K
16K-20K
12K-16K
8K-12K
4K-8K
0K-4K
X
X
X
X
7
X
5
X
X
X
3
4
0
6
1
2
} 虚页
页表
物理地址空间
页框
28K-32K
24K-28K
20K-24K
16K-20K
12K-16K
8K-12K
4K-8K
0K-4K

页表表项设计

- 页表由页表项组成
- 页框号、有效位、访问位、修改位、保护位
  - ✓ 页框号（内存块号、物理页面号、页帧号）
  - ✓ 有效位（驻留位、中断位）：表示该页是在内存还是在磁盘
  - ✓ 访问位：引用位
  - ✓ 修改位：此页在内存中是否被修改过
  - ✓ 保护位：读/可读写

通常，页表项是硬件设计的

关于页表

多级页表

- 32位虚拟地址空间的页表规模？

  页面大小为4K；页表项大小为4字节

  则：一个进程地址空间有 ？页 2^{20}

  其页表需要占 ？页（页表页） 1024

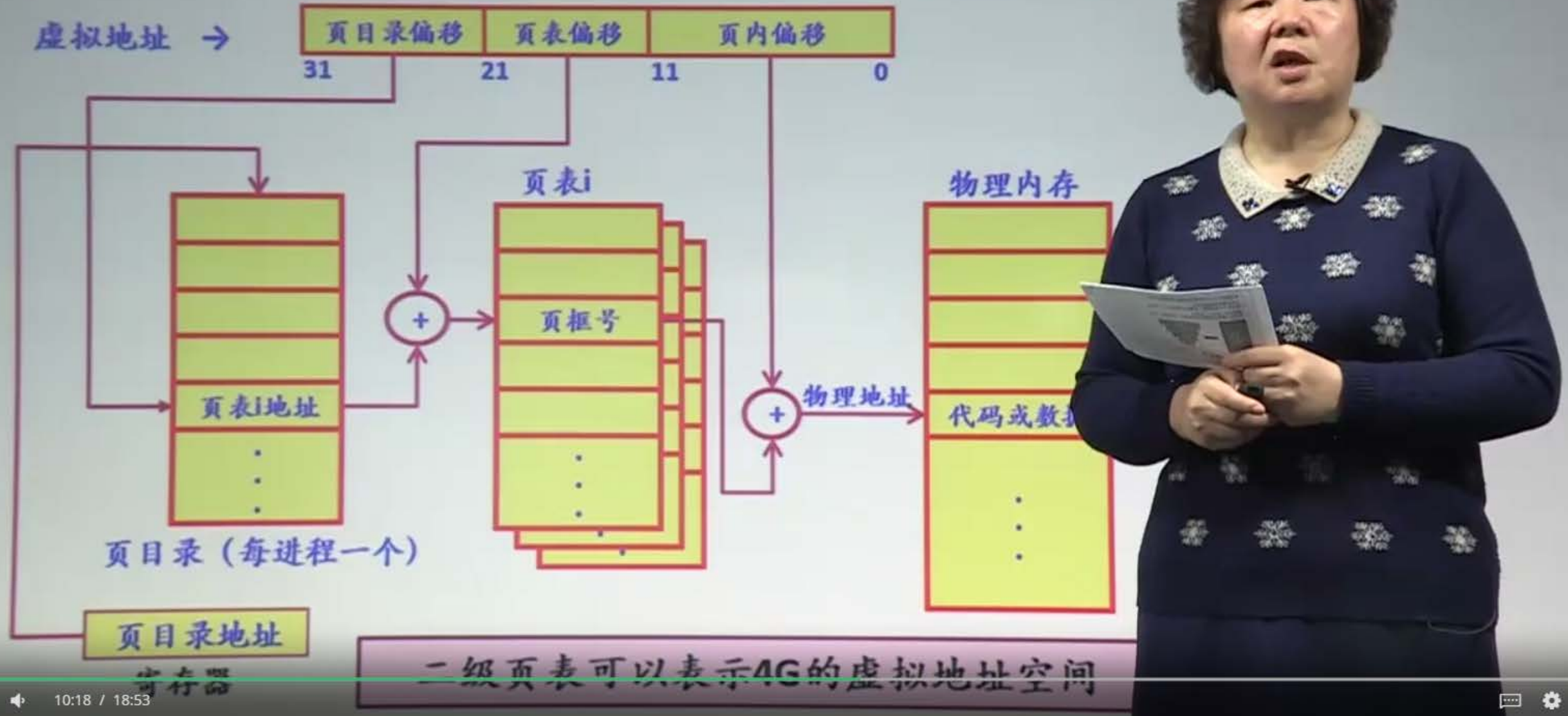
- 64位虚拟地址空间

  页面大小为4K；页表项大小为8字节

  页表规模： 32,000 TB

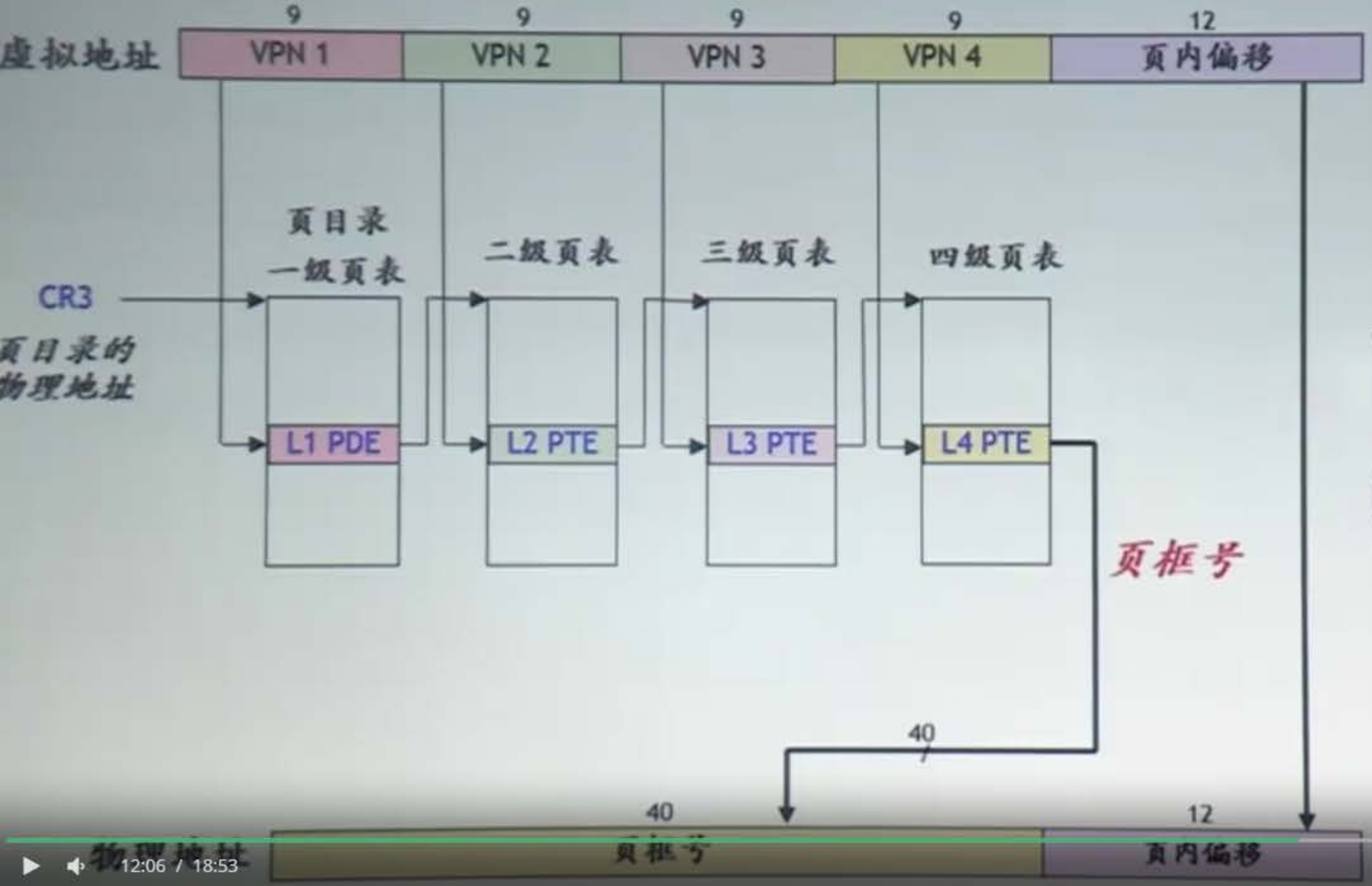
- 页表页在内存中若不连续存放，则需要引入页表页的地址索引表 → 页目录（Page Directory）

二级页表结构及地址映射



CORE I7 页表结构

虚拟地址空间2^{48}



I386页目录项和页表项

页目录项 PDE (Page Directory Entry)

PFN	Avail	G	PS	0	A	PCD	PWT	U/S	R/W	P

页表项 PTE (Page Table Entry)

PFN	Avail	G	0	D	A	PCD	PWT	U/S	R/W	P

PFN(Page Frame Number)：页框号
P(Present)：有效位
A(Accessed)：访问位
D(Dirty)：修改位
R/W(Read/Write)：只读/可读写
U/S(User/Supervisor)：用户/内核
PWT(Page Write Through)：缓存写策略
PCD(Page Cache Disable)：禁止缓存
PS(Page Size)：大页4M

引入反转(倒排)页表

- 地址转换

  从虚拟地址空间出发：虚拟地址 → 查页表 → 得到页框号 → 形成物理地址

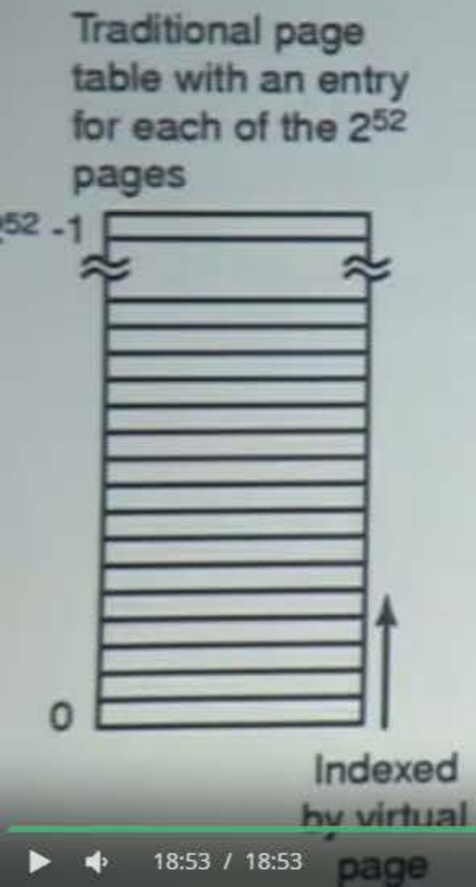
  每个进程一张页表

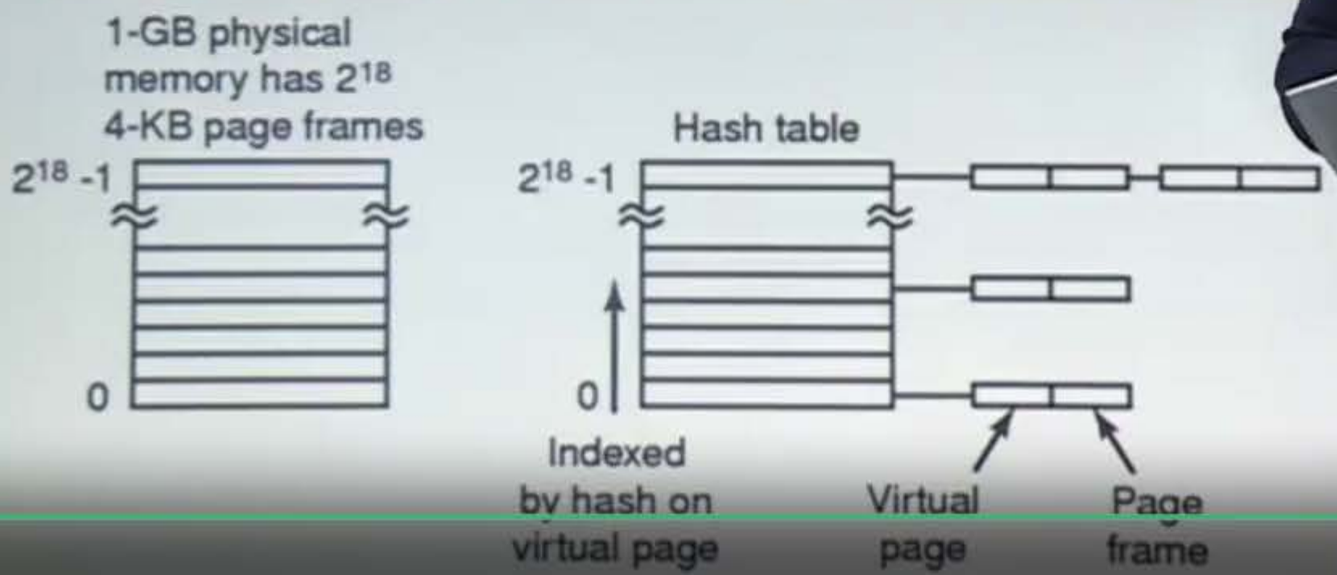
- 解决思路
  - 从物理地址空间出发，系统建立一张页表
  - 页表项记录进程i的某虚拟地址(虚页号)与页框号的映射关系

那么这就是引入反转页表的一个主要的理由

反转(倒排)页表设计

- PowerPC、UltraSPARC和IA-64 等体系结构采用
- 将虚拟地址的页号部分映射到一个散列值
- 散列值指向一个反转页表
- 反转页表大小与实际内存成固定比例，与进程个数无关





18:53 / 18:53

0:00 下面我们介绍，在虚拟页式存储管理方案当中 页表及页表项的设计问题 我们首先来看一张图 这张图当中左边是 虚拟地址空间，右边呢是物理地址空间 虚拟地址空间呢可以划分成若干个虚拟页面 而物理地址空间由若干个页框组成 那么我们可以看到，在这个示意当中 并不是虚拟地址空间的所有的虚页面都 加载到内存了，只是部分内容加载到内存 我们可以看到，比如说第 0 页加载到了页框 2 这个地方 第 1 页加载到了页框 1 那么我们这里头加载了大约七八个页面 那么我们当这个进程运行的过程中 如果要访问到了一些页面，那么就要 通过一个页表，由页表记录了哪些 页面已经加载到内存了，哪些页面没有加载到内存 那么通过页表的设计 页表的设置就可以得到这样的信息 那么当然了，页表是由页表项组成的 我们来看一下页表项的设计 通常情况下，在页表项当中应该保存这样一些信息 页框号、有效位、 访问位、 修改位和保护位 那么页框号是 最重要的一个信息。 那么我们把虚页号 对应到了哪一个具体的页框呢？那么就通过页框号来给出这个具体的 物理页面。 那么有效位呢是 确保这个页表项所对应的虚页面 它是在内存呢还是在磁盘上？ 通常有效位如果是 0，就表示这个页面还没有读进内存 那么这时候呢这个页框号其实是无效的 如果有效位为 1 ，就表示这个 相应的虚页面的内容已经读入内存了，那么这个页框号就是有效的好，那么下面还有两个重要的位，一个是访问位 当要读这个页面，或者使用这个页面的时候 那么由硬件将这一位设置成 1 表示这个页面，在内存的这个页框内容被访问过 那么另外一个是修改位，也就是说 当这个在内存的页框当中的内容 被修改了，那么由硬件自动地将这个位设置成 1 当然我们知道，如果在内存期间被修改了 那么将来还要把它写回到磁盘。 所以这一位实际上是 用于这样一个目的。 那么一个页面在内存 那么我们对它可能有一些权限的设置，比如说这个页面是只读页面 还是这个页面可以进行读写操作，那么就通过 保护位来设置，通过保护位的设置来达到这样一个要求 这就是页表项的一个基本的设计。 当然了，一般的实际的系统中 页表项会，还有很多的内容，待会儿我们再介绍 那么页表项是谁来决定的呢？ 通常情况下，由于在地址转换过程中 是由硬件来完成这个转换，那么硬件需要去访问页表，因此 这个页表项的设计通常是由硬件设计的 这只是一个一般情况，但是有些计算机可能 不是这样，这不是一个必须的，大部分的硬件是这么做的 那么下面我们来看看页表 我们来问大家这样一个 问题，如果我是一个 32 位的地址空间，那么它的页表 有多大规模呢？那么如果页面大小是 4K 页表项呢我用四字节来表示的话 那么一个进程的地址空间呢需要占多少页面呢？ 那我们可以看到，需要 2 的 20 次方个页面 一个

页面就有一个页表项 于是我就有 2 的 20 次方个页表项 那么我们的页面大小是 4K 页表项是 4 字节，也就是一个页面可以放 1K 个 页表项。 我们有这么多个页表项，我们需要放 多少页面，才能够把页表存放进来呢？ 所以我们来看一下，页表页，就是这个内容是 页表，所以我们把它称之为页表页。 那么页表页它需要多少呢？需要 1024 个页表页 那么这是一个进程，也就是每个进程都需要 1024 个页表页 如果我有 100 个进程，大家就可以算出来有多少个页表页 那这也是要占据内存空间的 这是 32 位计算机，那么如果我们换成 64 位的计算机呢？ 那么 64 位的虚地址空间，那么页面大小假设还是 4K 为了表示这么大的一个地址空间，我们的页表项就需要 8 个字节了 那在这种情况下，页表本身的规模 一个进程的页表本身的规模就要达到 3 万多的 TB 这是个非常庞大的数字，所以我们知道 作为页表，它保存在内存 不应该连续存放在内存，所以我们这里头 把页表页在内存的位置让它不连续存放 那么如果不连续存放，那么我们就需要引入一个新的页表页来保存这些 页表页的地址，那么这个呢，我们就称之为页目录，页目录 这样的话呢，我们一个页表就变成了，原来是一个一维 的页表的话，那么就需要，从这样一个角度就变成了一个二维的 那么甚至呢是一个多级的页表 所以现在大部分的这个计算机系统，都是提供的是多级页表这样一种结构 我们来看一下简单的啊，二级页表结构以及 在这样一个页表结构下它的地址转换过程的示意 我们来看一下刚才所说要有 32 位的 地址空间，有若干个页表页，那么有 1024 个页表页 那么我们这里头有一个页目录，页目录。 这样的话每个进程有这么一个页目录 在页目录当中的每一行，那么页目录项当中，那么就保存了页表的地址 所以先查找页目录，就得到了页表的地址 然后再从页表当中的页表项当中呢，我们通过页框号呢才能够取 形成真正的物理的内存地址 那么页目录，当一个进程上 CPU 的时候 那它的页目录的起始地址，应该推送到一个特定的寄存器里头去 在 X86 的这个体系结构当中呢，这个寄存器呢是 CL3 好，那么这个寄存器存放了页目录的起始地址 那么当这个进程下 CPU 的时候，那么这个地址呢，就会 保存在这个进程的 PCB 相关的现场信息里头 通过这样一个页目录的地址我们就可以找到内存中的页目录 好，那么我们来看看地址转换过程，一个虚拟地址 那么虚拟地址呢，是 32 位的虚拟地址空间，那么 页面大小是 4K，所以我们的页内偏移呢，是占 12 位 前面的页框号呢，我们把它一分为二啊，分为 10 位和 10 位 前面 10 位呢用于页目录的索引或者是 代表了它对页目录的一个偏移的一个位

置 所以我们通过前面这 10 位找到页目录对应的页目录项 找到页目录项之后就得到了页表的地址，然后我们再通过页表的偏移和 页目录的这个当中，页表的地址，我们就可以 计算出来页表项的位置 那么页表项的位置有了以后，我们就有了页框号，那么我们就可以通过这个页框号以及 页内偏移经过一个拼接，就形成了真正要访问到的内存的物理地址 或者是代码的这个相应的位置。 那这就是 二级页表啊，以及它的地址转换的一个过程的示意 那我们来看到，在二级页表的情况下，那么 最大二级页表可以表示的这个，虚拟地址空间呢就是4G 4G。 如果超过了 4G，那么二级页表肯定是不够的，我们来看一下 CORE I7 的页表结构，我们可以看到CORE I7的页表结构呢，是四级页表，啊，四级页表 那么，它的虚拟地址多大呢？我们可以看到是 2的48次方，虚拟地址的这个空间呢是2的48次方 页内偏移还占12位，那么48减去12，36。 然后我们可以 看到把36位分解成4个部分，每部分占9位 那么页表分成几级呢？四级，第一级就是我们说的页目录 然后是二级页表，三级页表和四级页表，每一级 页表的索引，都是通过相应的这9位来进行索引的 那么我们一样有一个CR3，或者说一个页目录的 起始地址的这个寄存器，存放了页目录的起始地址 通过这样一个寄存器，我们就可以找到页目录，那么地址转换的过程呢，就是首先用 第一级的这9位，那么我们把它称之为序列号1的话，那么来查页目录 通过查页目录呢，得到了二级的页表的起始地址，那么 二级页表呢，又通过这个我们所说的中间这个9位，来做索引，然后 是三级页表，然后是四级页表，所以每一级页表有 页表项，当然第一级一般我们通常称为页目录 那么最后到了四级页表，就查到了页框号 那么，这个页框号就和页内偏移拼接成了一个物理地址 啊，这里我们可以看到，这是一个2的48次方的，这么大的一个地址空间 那么关于页目录项和页表项，我们来看一下 它的具体内容，那我们举的例子呢是，I386这样一个体系结构当中 页目录项和页表项的内容，我们主要看一下页表项，啊页表项，那这里头我们可以看到 这32位，就是4字节的页表项当中，最右边这一位 P位，实际上是什么呢？有效位，由这一位来 决定这一个页位面是进内存了，还是没有进内存 如果没有进内存，就会引发一个 异常发生，那么 第2位是R/W，R/W实际上就是读写位 就是表示这个页面是只读，还是可读写 U/S位呢，实际上就是说 确定这个页面啊，是操作系统访问，普通用户 不能访问，那么就相当于保护这个页面，只能操作系统访问，只能内核访问 就是用U/S这一位。 那么还有 这个A位，就是我们前面也介绍

过，叫引用位或者是存取位，访问位 只要是这个页面被访问了，那么硬件就会把它设置 成1，那D位，叫Dirty，实际上就是我们前面的修改位 也叫脏位，实际上就是说，这个内容被修改过了 一旦做了写操作就由硬件把它设置成1 这里头还有一些位，比如说PCD这个位，是禁止缓存，也就是说这块儿 这一部分页面的内容不能够送入缓存里头去，要禁止缓存位，如果 允许缓存的话，那么我们可能有一个写入的方式，写入的 策略，是由PWT来决定，是回写的方式，还是通写的方式 PS位呢，指的是大页，啊，大页，页面大小 那么，通常情况下呢，我们采用的是4K大小的页面 但是，有的计算机系统呢，它还可以支持其它尺寸的页面 那么PS设置为1的话呢，实际上是表示的是大页面，在I386当中呢就指的是4M的页面 这是关于页表项和页目录项的一些位的介绍 那么下面我们来介绍反转页表或者叫倒排页表 那么这个问题呢是这样的，我们来看一下地址转换 那我们现在的地址转换呢，是从虚拟地址空间出发 啊，拿到了虚拟地址之后 硬件会去用虚拟地址的前面，页号部分去查页表 查完页表之后呢，最后得到了页框号 然后把它拼接成一个物理地址 因此呢，我们从虚拟地址空间出发的话呢，我们每个进程有一张页表 那么我们刚才也已经讲过，这个页表是非常巨大的，占据了很多的空间 即便是在内存当中，不连续存放，那么你也需要花很多的空间来 存放页表，那么怎么解决这样一个问题呢？ 我们的解决思路是，我们不从虚拟地址空间出发 我们从物理地址空间出发，也就是从物理内存出发，因为物理内存呢 大小相对是固定，相对稳定一段时间 所以我们从物理地址空间出发，整个系统就建立一张页表 而不是给每个进程建立一张页表，而这个页表当中呢 本身页表是为了进行地址转换所做的一个一个数据结构，那么所以这个 页表当中 一定要记录了虚拟地址和物理页框号的一个对应关系，那么我们现在这种新的 这种反转页表或者是倒排页表 它的每一个页表项，实际上是记录了某个进程，比如进程i 它的某个虚拟地址或者叫虚拟页号 与页框号的一个映射关系，那么它记录的信息是 这样一些信息，他要把进程的信息，也填写在页表项当中 那么这就是引入反转页表的一个主要的理由 那么在引入反转页表之后，我们来看一下 那么原来这么大的一个地址空间，每个进程一个 那么我们需要一个非常大的页表 而且是多张页表在内存，那么如果我们按照一个 物理内存大小来建立页表呢，那么这样呢 就会节省了很多空间，而且整个系统只有一张页表 但是呢，这里存在了其它的问题 首先我们来看看，像这样一种方案，实际上 是64位计算机通常会采用的方案，像PowerPC

UltraSPARC还有我们说的IA这个64位的这种体系结构 通常都会采用这种反转页表或者倒排页表 但是当一个虚拟地址，CPU取到了虚拟地址 需要去进行地址转换的时候呢，它要去把整个页表进行查找 那么这样的话呢，就会带来了一个新的问题，就会查整个页表，也会耗费不少的开销 因此呢通常呢，我们可以解决这个问题呢，是通过 一个Hash，Hash表来解决这个问题，也就是 我们可以将虚拟地址的页号部分，先映射到了一个散列值 然而这个散列值呢，实际上就是指向了反转页表当中的某个位置 那么当进行地址转换的时候呢，我们可以通过对虚拟页号 进行Hash，然后找到对应的这个反转页表的相应的页表项 当然这个过程当中呢，可能会有一些冲突 那么解决冲突的话，那么还是需要拉链这种方法，所以这张图呢给出了一个简单的示意 那么有了反转页表之后呢，因为我们知道反转页表的大小 实际上是和实际内存大小是固定比例的 那么就与进程个数无关，所以呢现在的计算机，新的计算机 体系结构呢，用这种方法来解决页表占用很大空间的问题