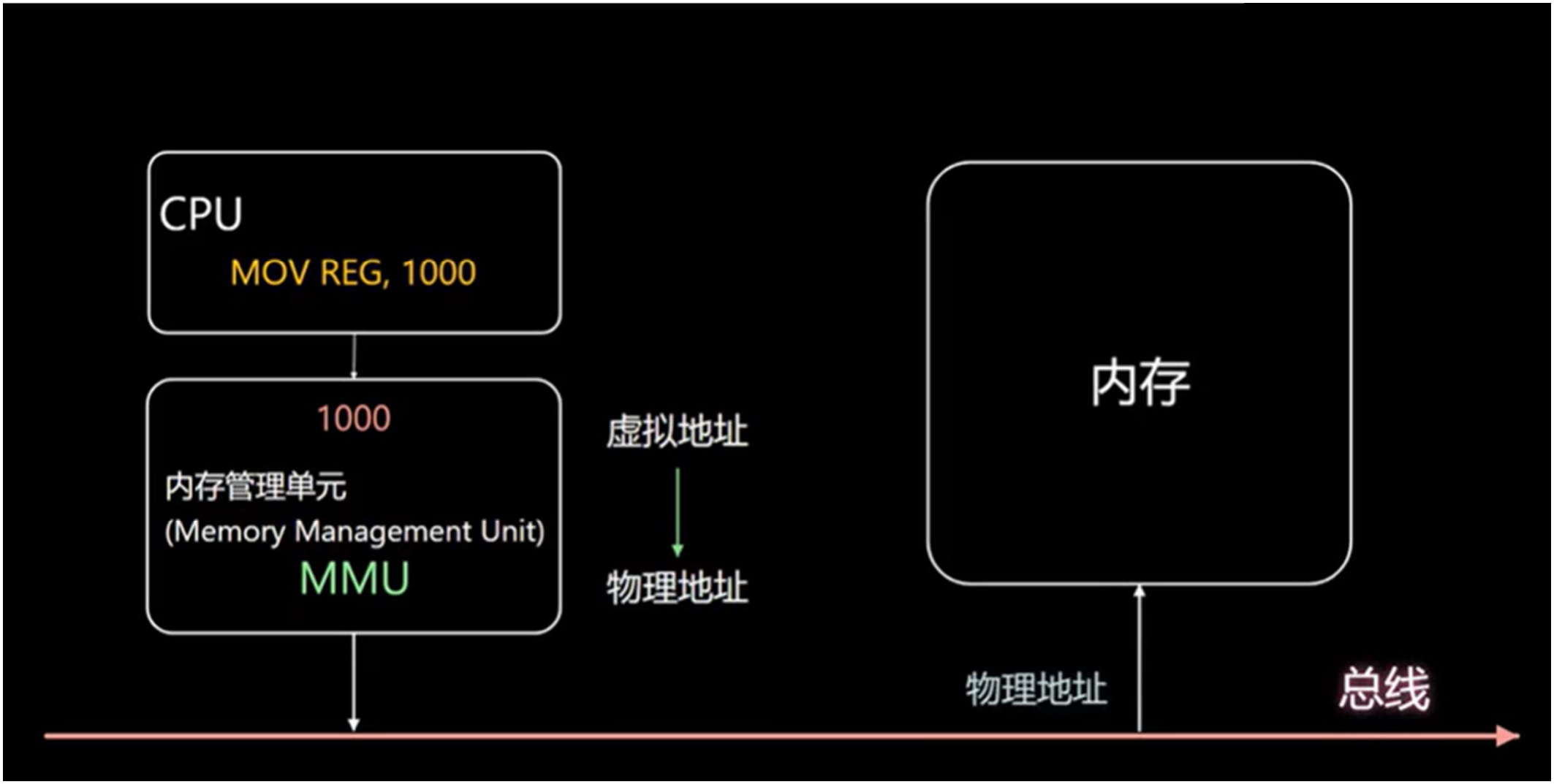
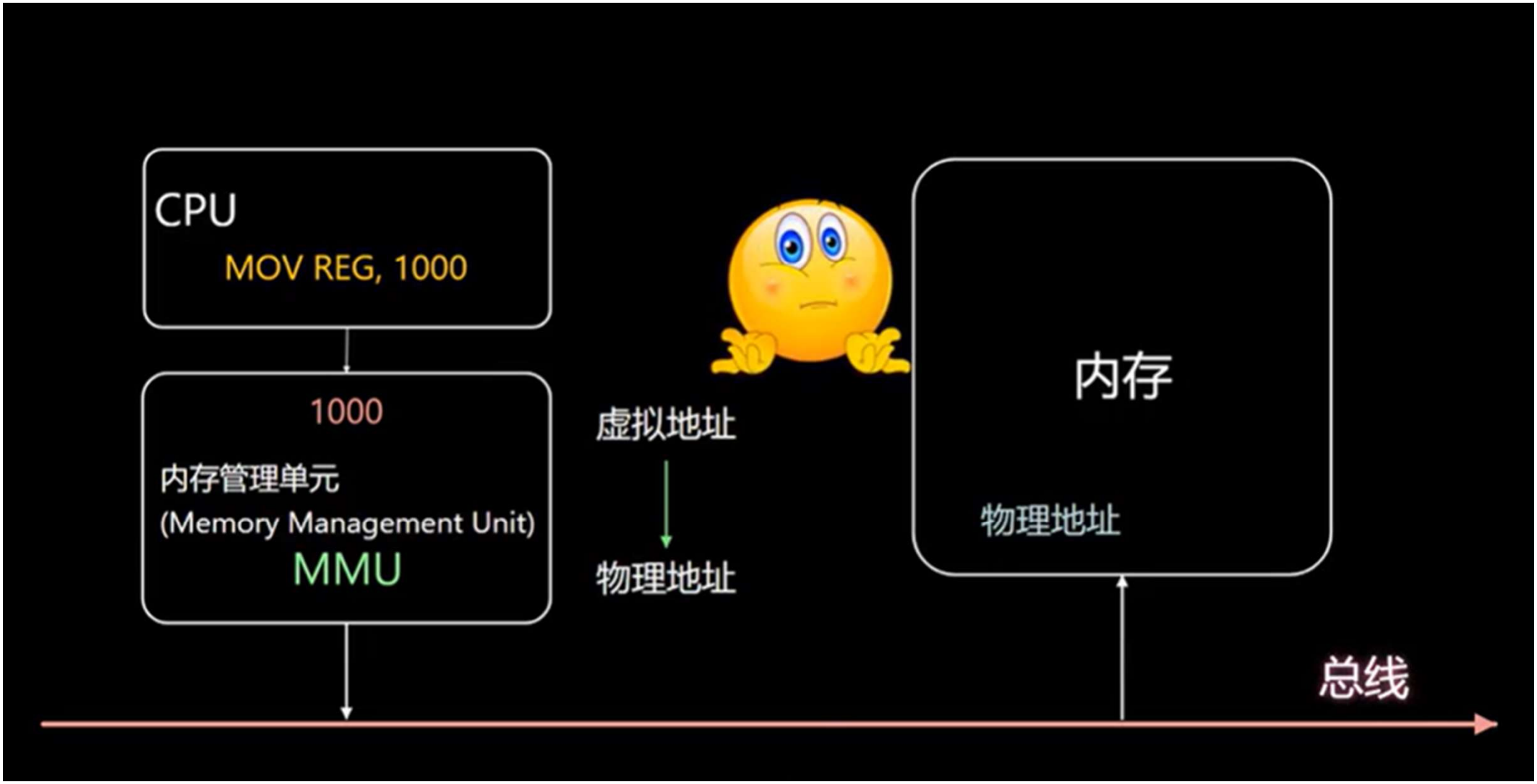
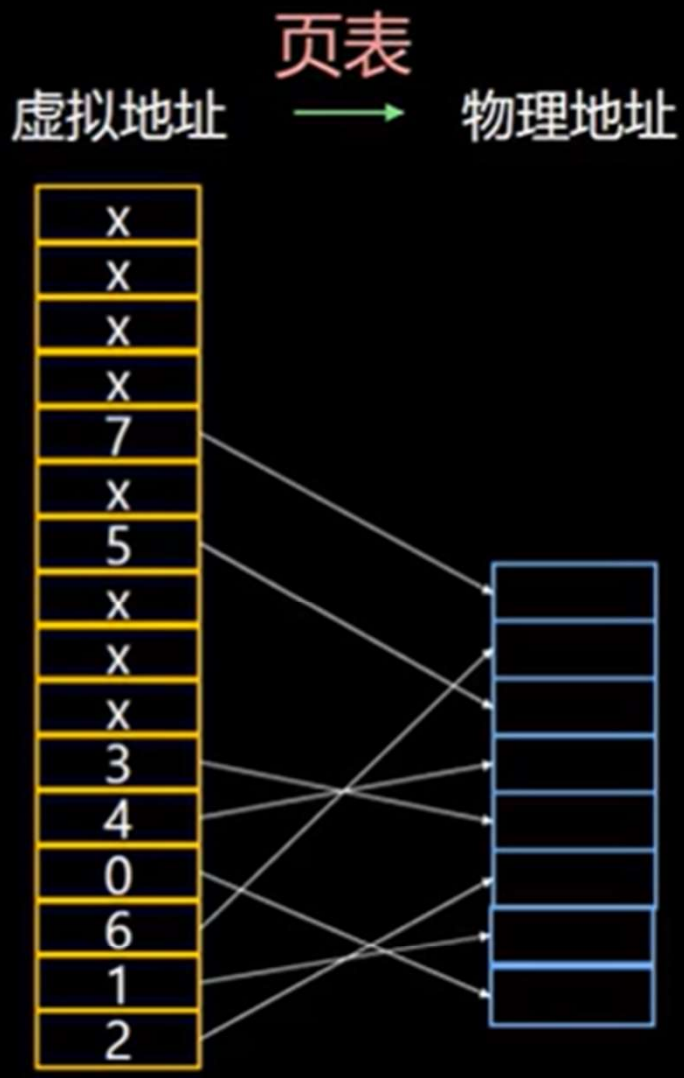

虚拟地址
MOV REG, 1000

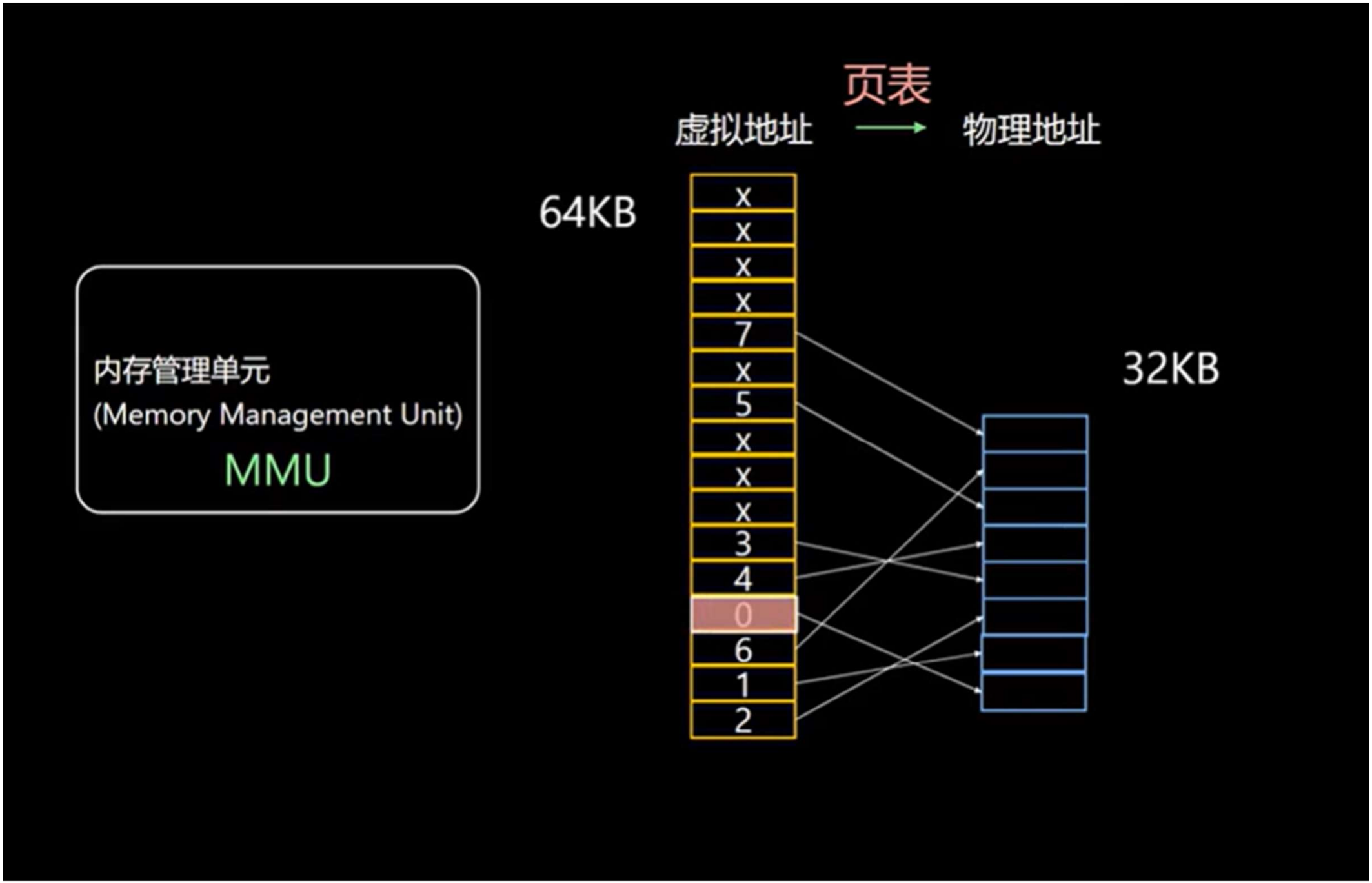

CPU
MOV REG, 1000
1000
内存管理单元
(Memory Management Unit)
MMU
虚拟地址
物理地址
总线

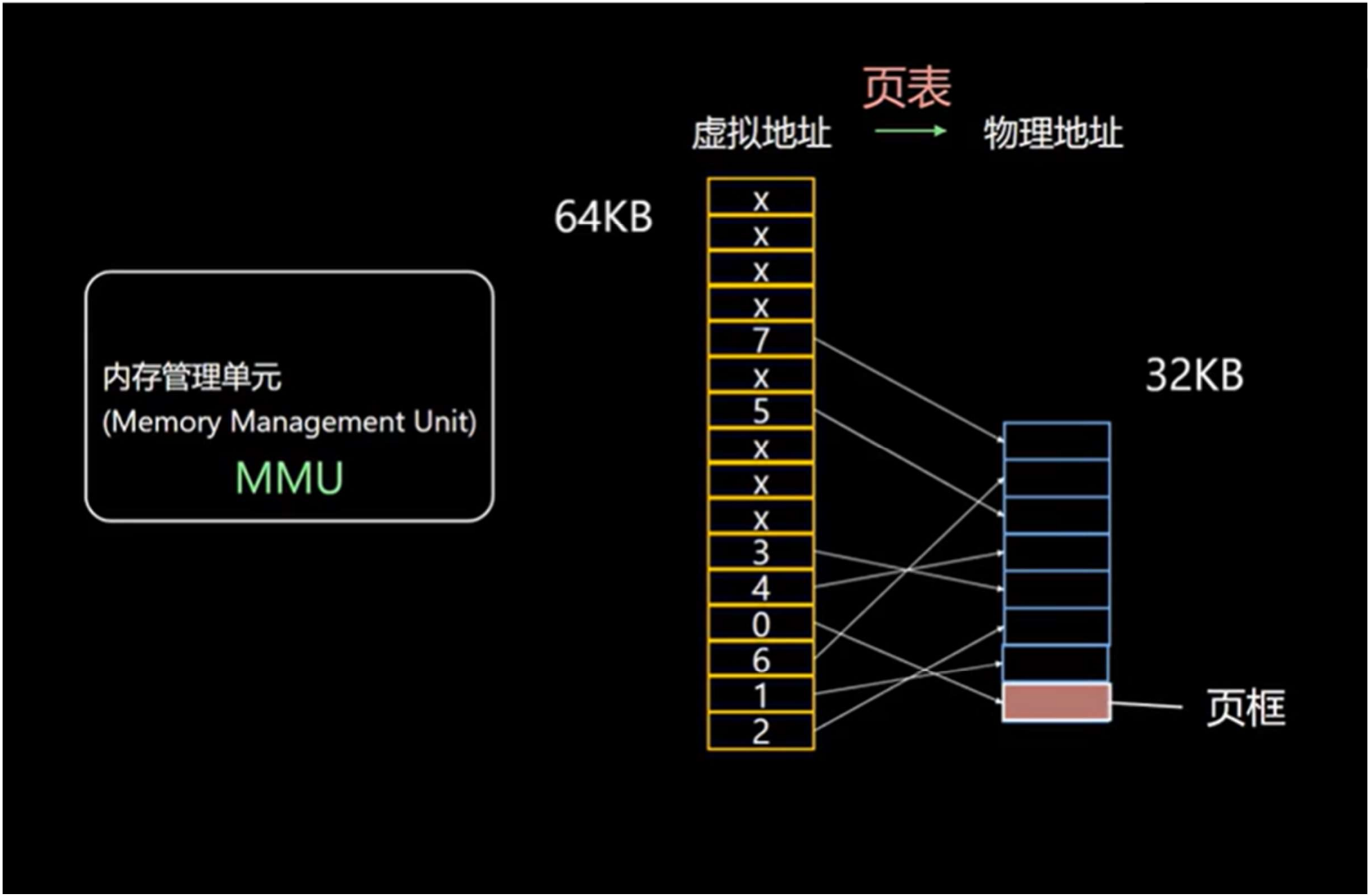

CPU
MOV REG, 1000
1000
内存管理单元
(Memory Management Unit)
MMU
虚拟地址
物理地址
内存
物理地址
总线

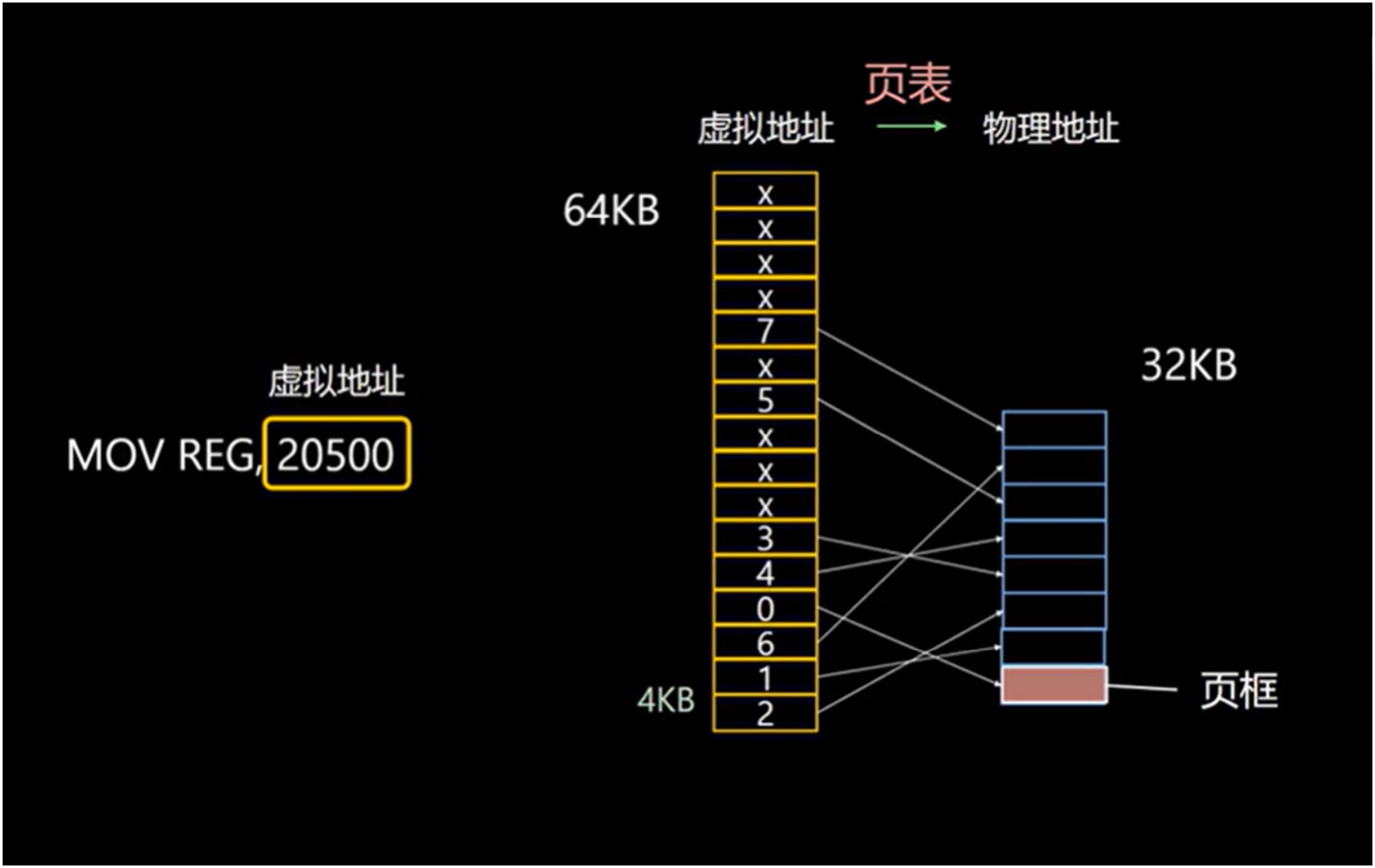

CPU
MOV REG, 1000
1000
内存管理单元
(Memory Management Unit)
MMU
虚拟地址
物理地址
内存
物理地址
总线

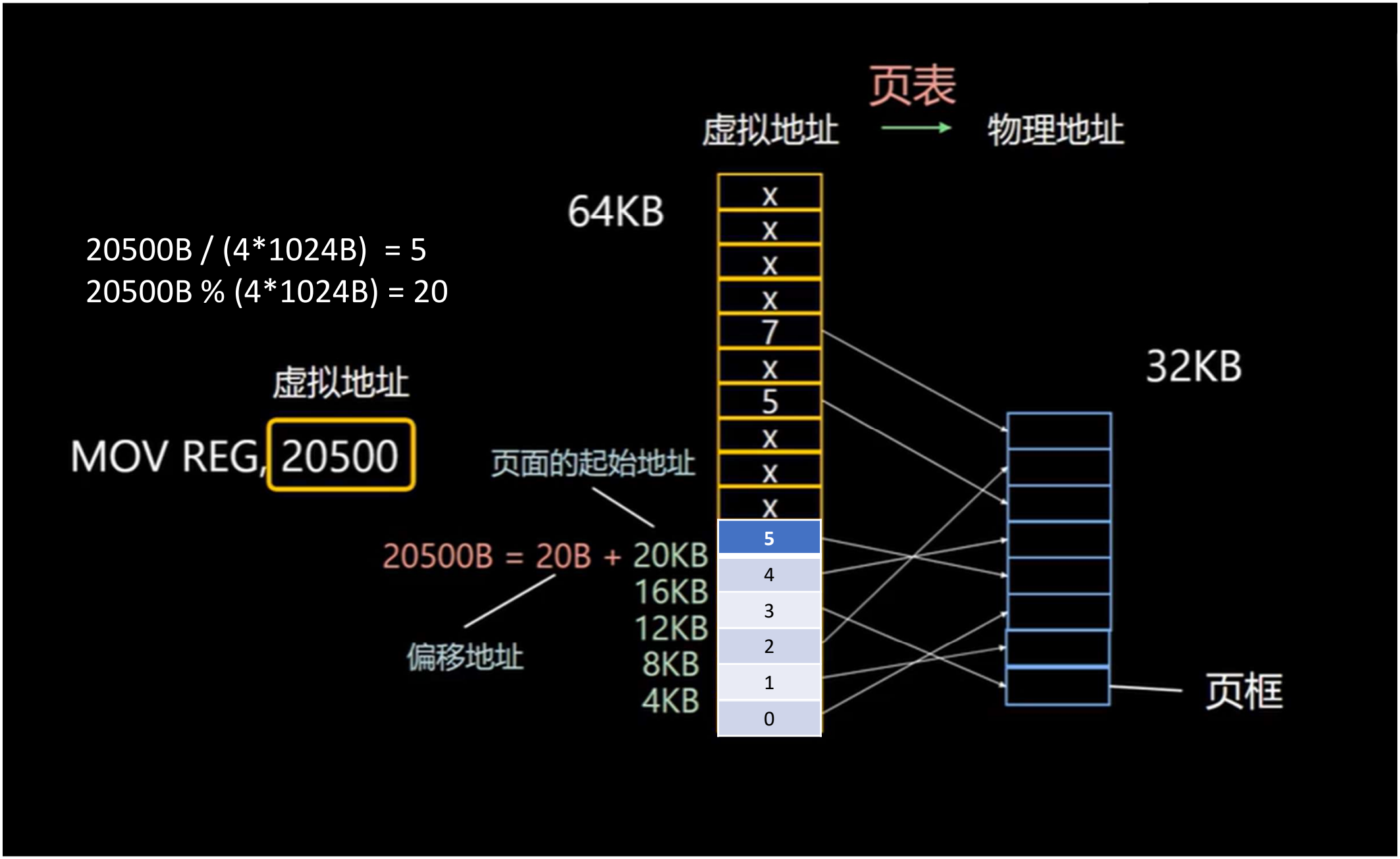
内存管理单元
(Memory Management Unit)
MMU

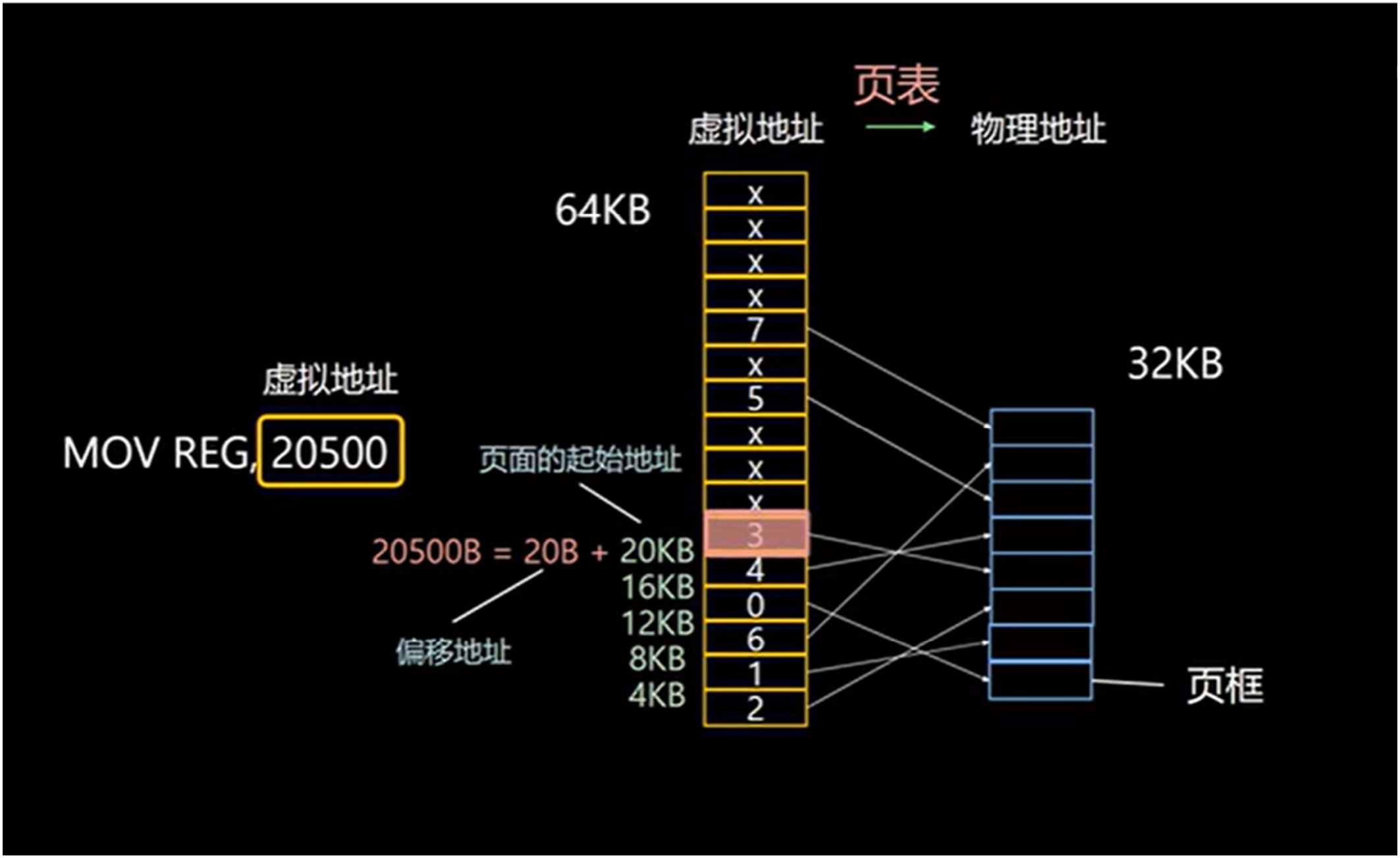


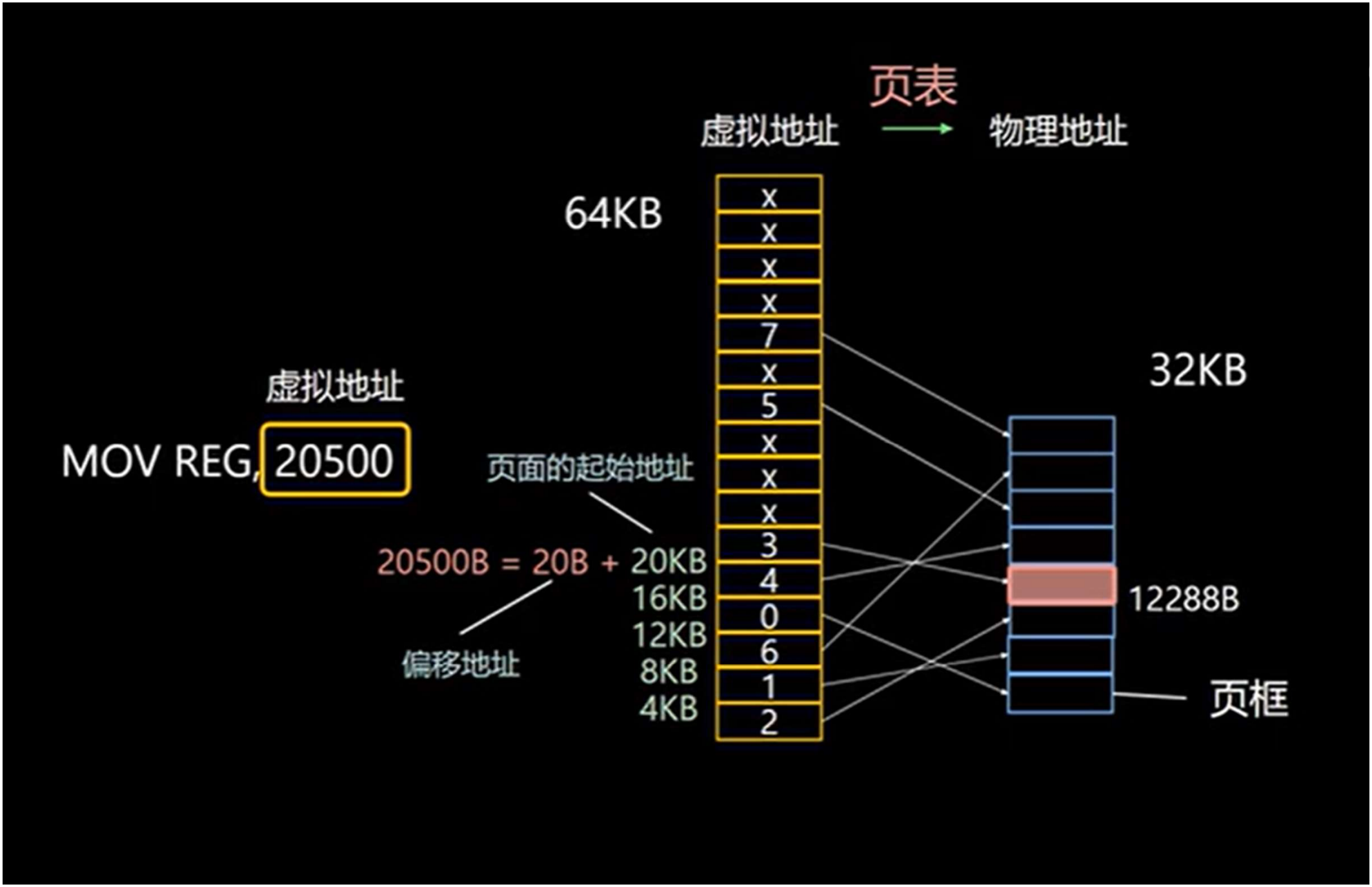

页表
虚拟地址
物理地址
64KB
32KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
内存管理单元
(Memory Management Unit)
MMU

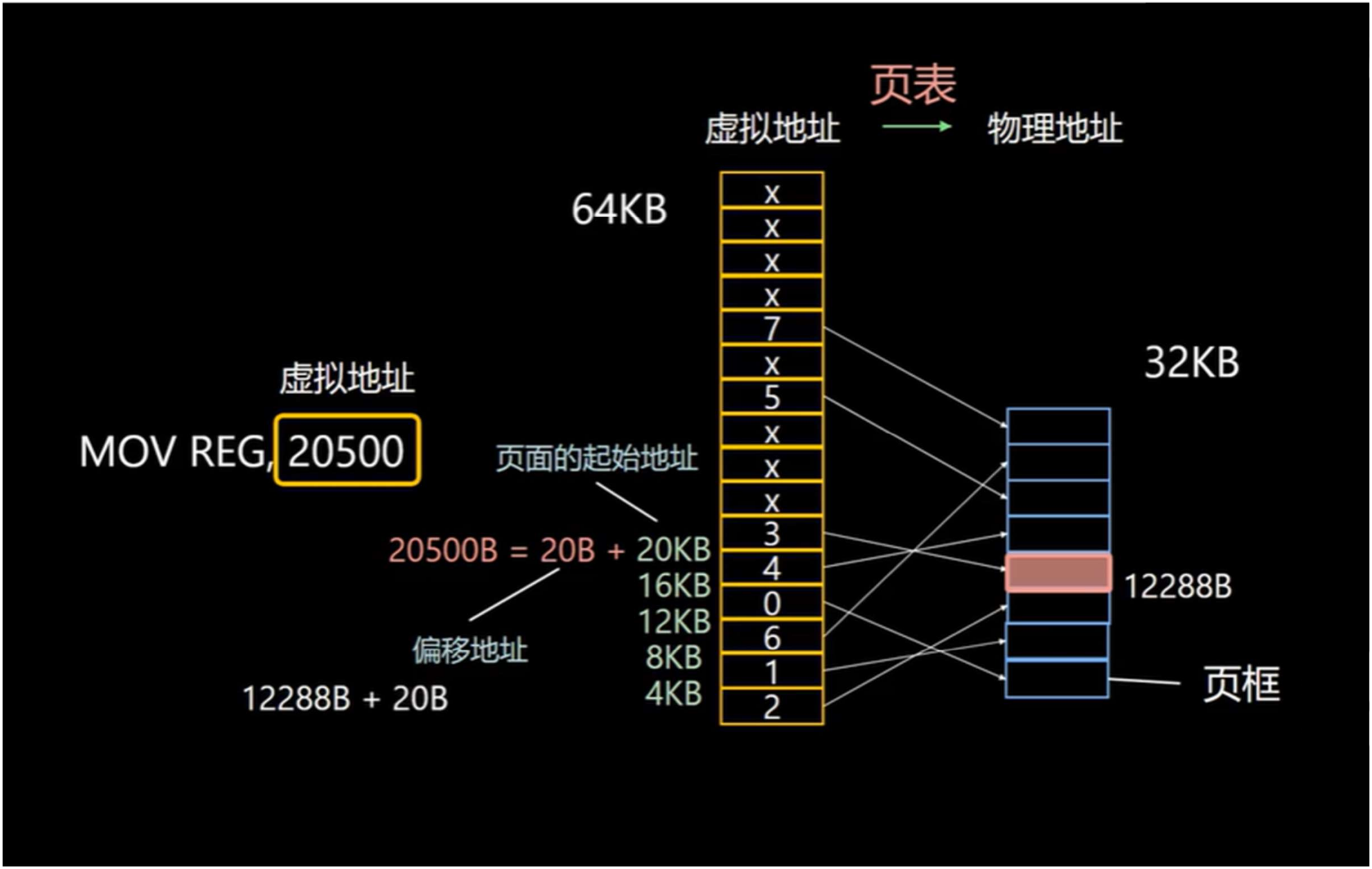

页表
虚拟地址
物理地址
64KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
32KB
内存管理单元
(Memory Management Unit)
MMU
页框

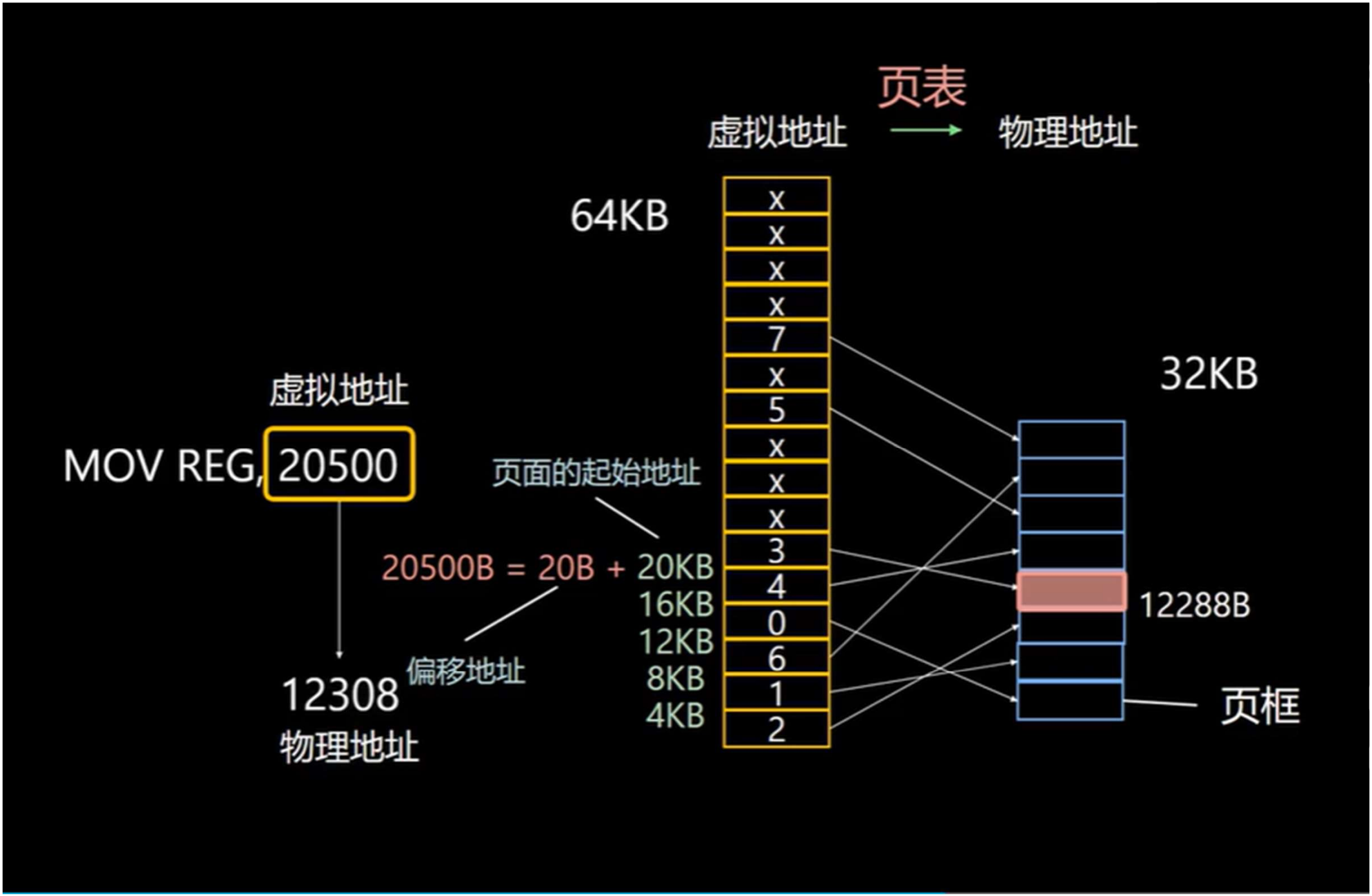

页表
虚拟地址
物理地址
64KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
4KB
32KB
页框
虚拟地址
MOV REG, 20500


页表
虚拟地址
物理地址
64KB
20500B / (4*1024B) = 5
20500B % (4*1024B) = 20
虚拟地址
MOV REG, 20500
页面的起始地址
20500B = 20B + 20KB
偏移地址
16KB
12KB
8KB
4KB
x
x
x
x
7
x
5
x
x
x
5
4
3
2
1
0
32KB
页框


页表
虚拟地址
物理地址
64KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
32KB
虚拟地址
MOV REG, 20500
页面的起始地址
20500B = 20B + 20KB
偏移地址
16KB
12KB
8KB
4KB
页框


页表
虚拟地址
物理地址
64KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
32KB
12288B
页框
虚拟地址
MOV REG, 20500
页面的起始地址
20500B = 20B + 20KB
偏移地址
16KB
12KB
8KB
4KB


页表
虚拟地址
物理地址
64KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
32KB
虚拟地址
MOV REG, 20500
页面的起始地址
20500B = 20B + 20KB
16KB
12KB
8KB
4KB
偏移地址
12288B + 20B
12288B
页框

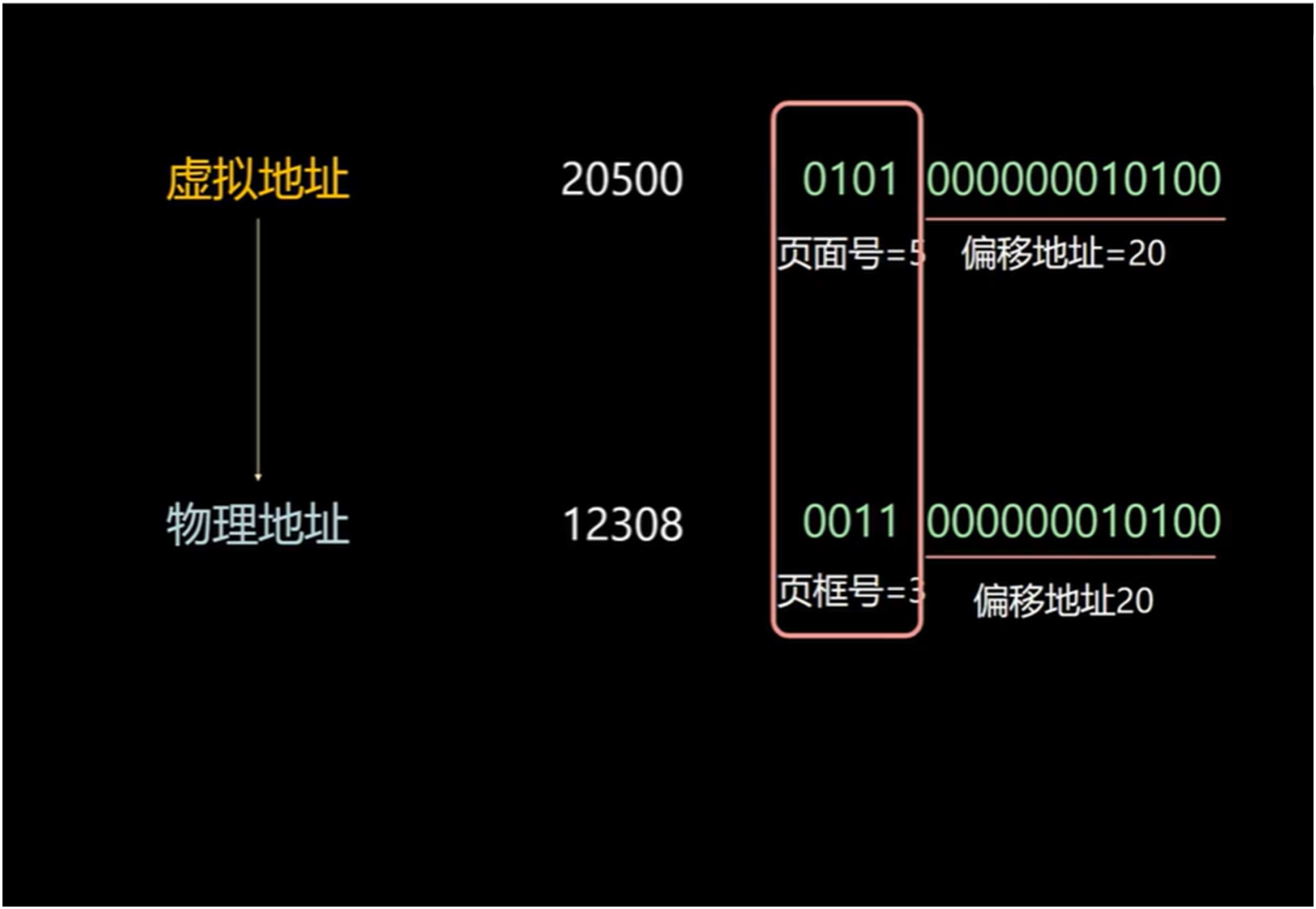

页表
虚拟地址
物理地址
64KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
32KB
12288B
页框
虚拟地址
MOV REG, 20500
页面的起始地址
20500B = 20B + 20KB
16KB
12KB
8KB
4KB
偏移地址
12308
物理地址

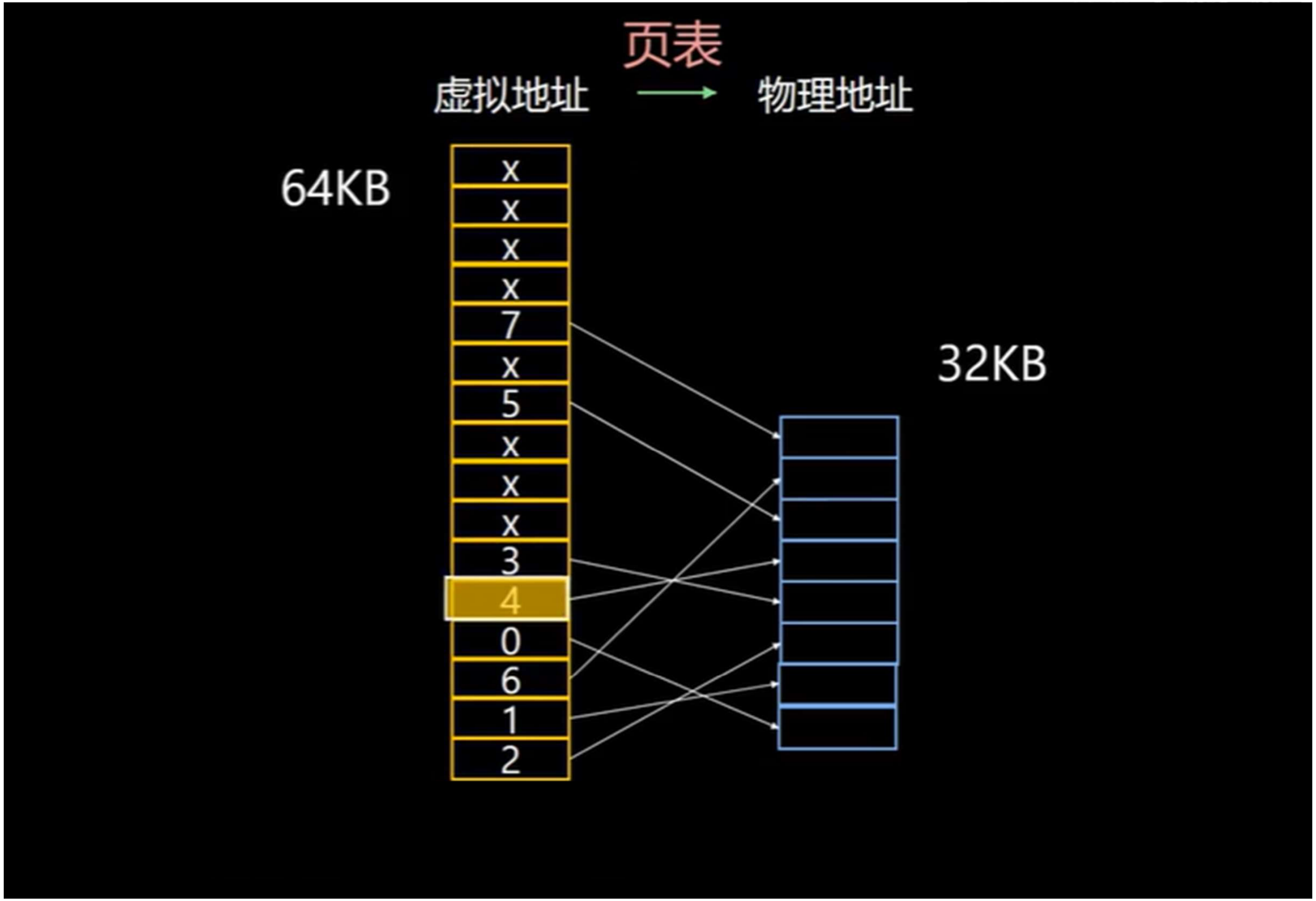
虚拟地址　20500　0101 000000010100

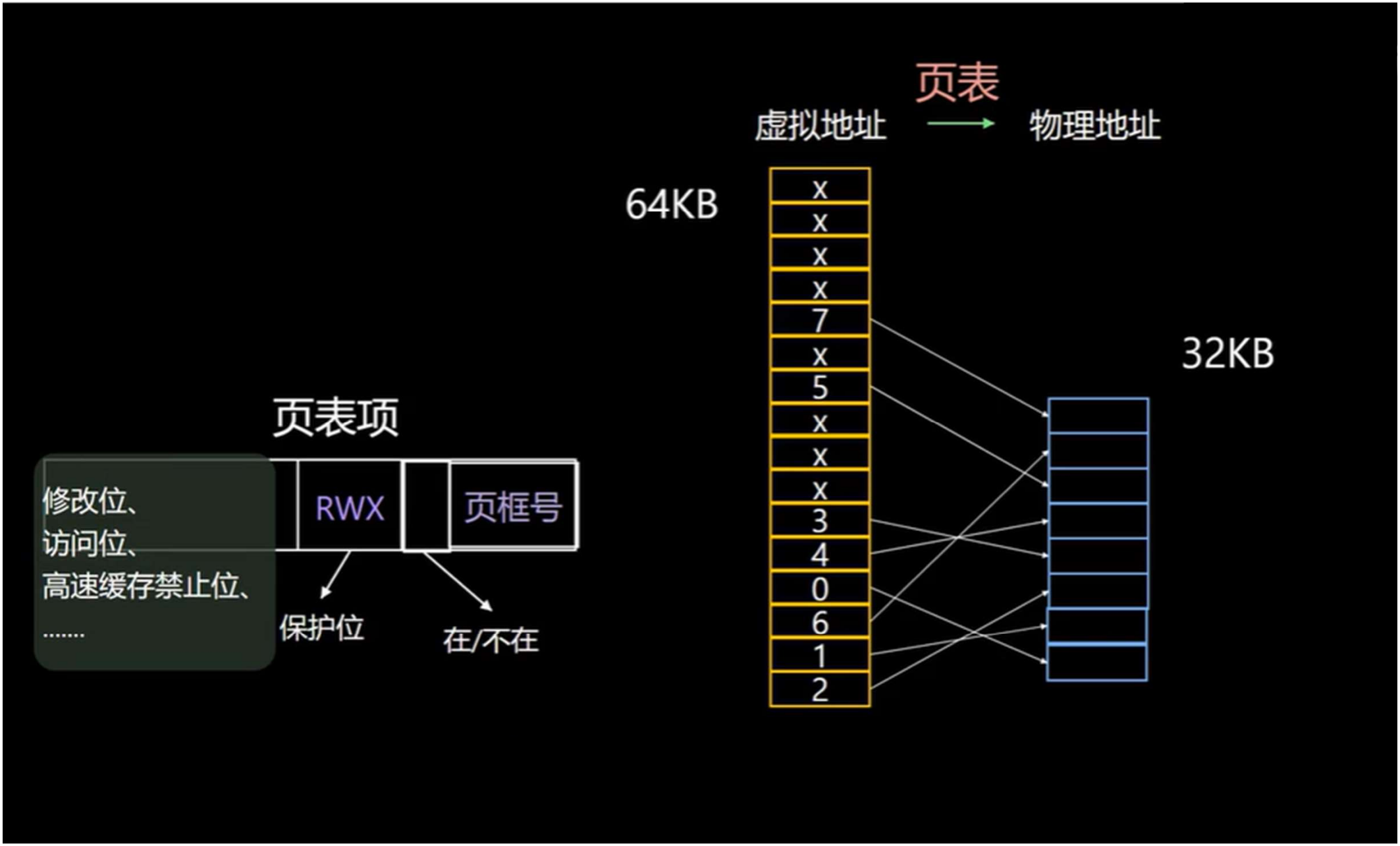
页面号=5　偏移地址=20

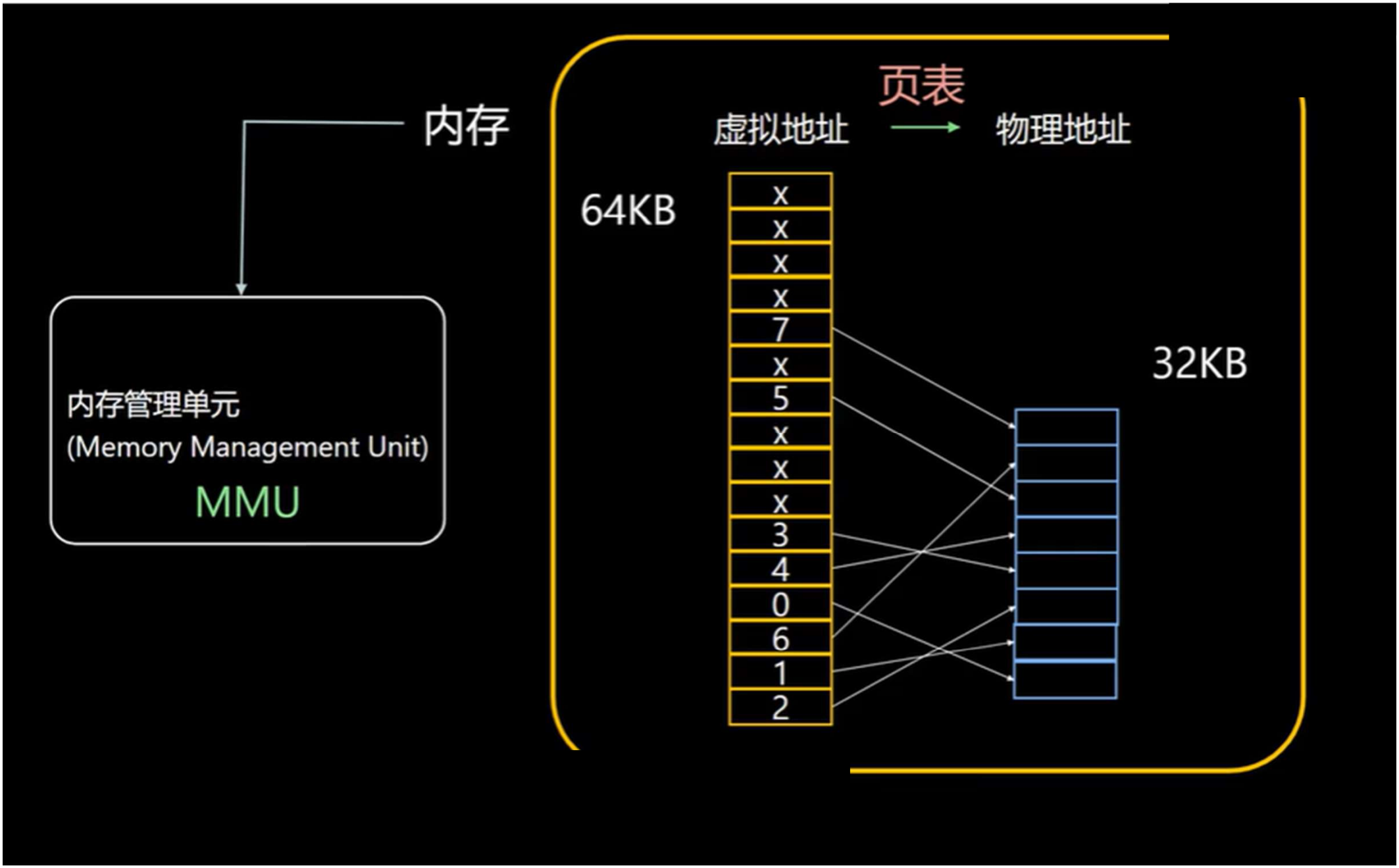
物理地址　12308　0011 000000010100

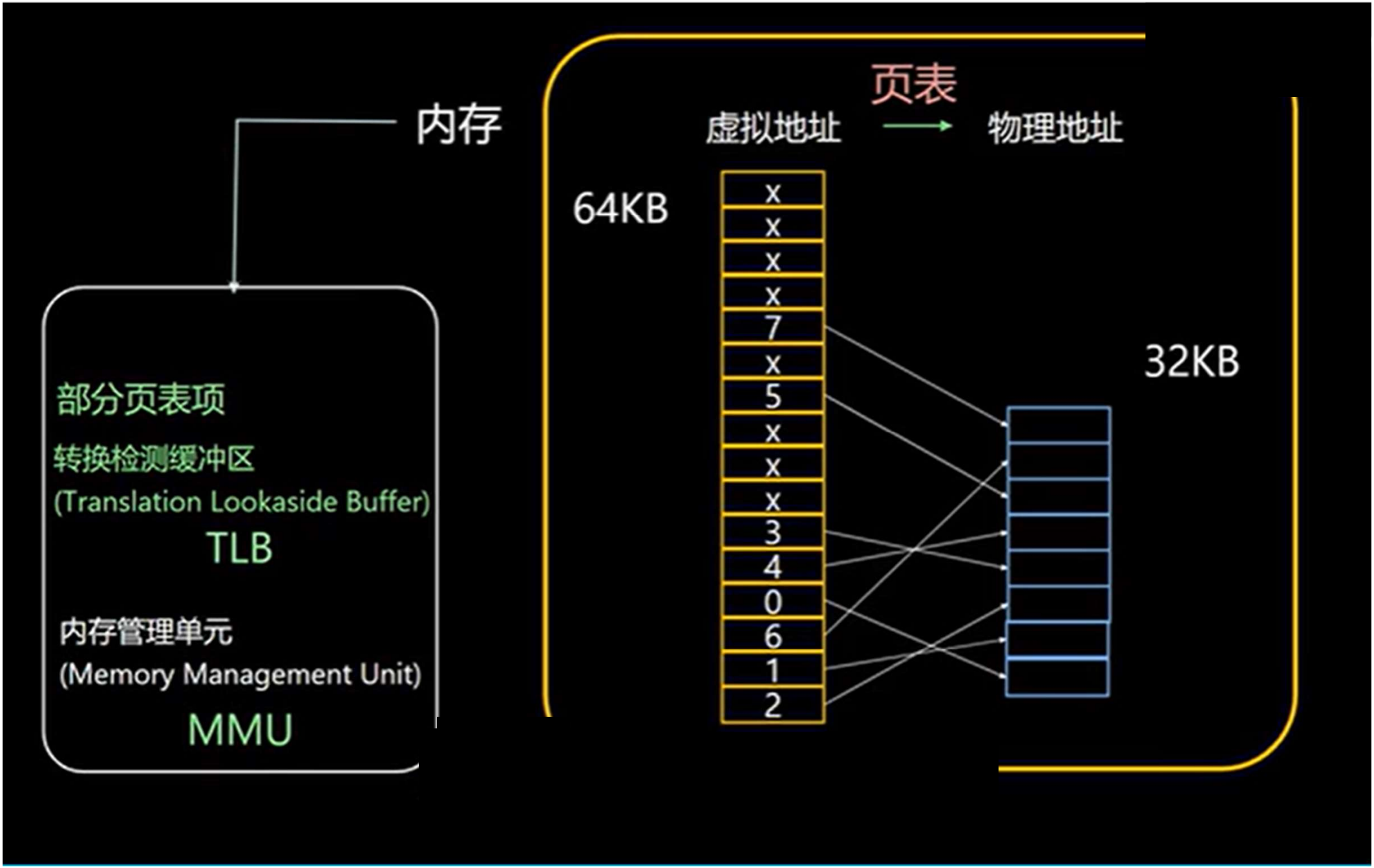
页框号=3　偏移地址20


虚拟地址
20500
0101
000000010100
页面号=5
偏移地址=20
物理地址
12308
0011
000000010100
页框号=3
偏移地址20


页表
虚拟地址
物理地址
64KB
32KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2


页表
虚拟地址
物理地址
64KB
32KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
页表项
修改位、
访问位、
高速缓存禁止位、
.....
RWX
页框号
保护位
在/不在


内存
内存管理单元
(Memory Management Unit)
MMU
页表
虚拟地址
物理地址
64KB
32KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2


内存
部分页表项
转换检测缓冲区
(Translation Lookaside Buffer)
TLB
内存管理单元
(Memory Management Unit)
MMU
页表
虚拟地址
物理地址
64KB
x
x
x
x
7
x
5
x
x
x
3
4
0
6
1
2
32KB

分段存储管理机制

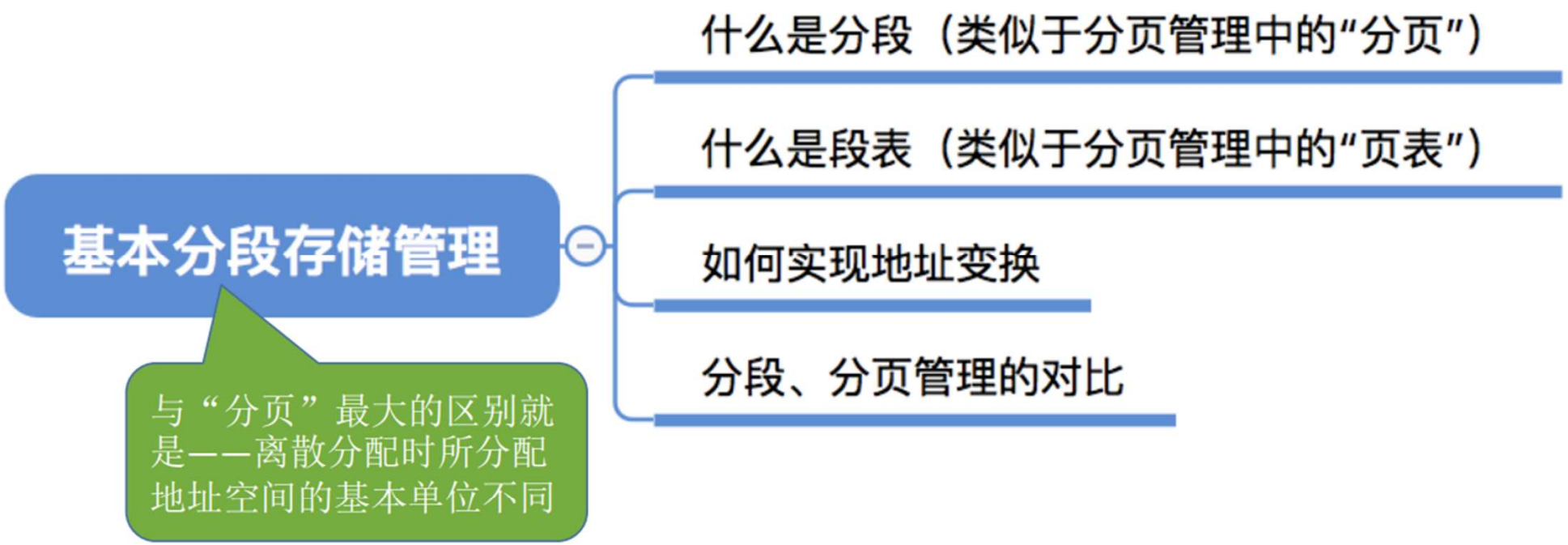
Linux
Android
Linux
OpenStack
Mac OS
Windows

分段存储管理机制

本讲内容

1. 逻辑分段与内存划分
2. 分段存储的管理表格
3. 分段存储的地址转换
4. 分页和分段存储比较
5. 段页式存储管理方案

知识总览


基本分段存储管理
与“分页”最大的区别就是——离散分配时所分配地址空间的基本单位不同
什么是分段（类似于分页管理中的“分页”）
什么是段表（类似于分页管理中的“页表”）
如何实现地址变换
分段、分页管理的对比

逻辑分段与内存划分

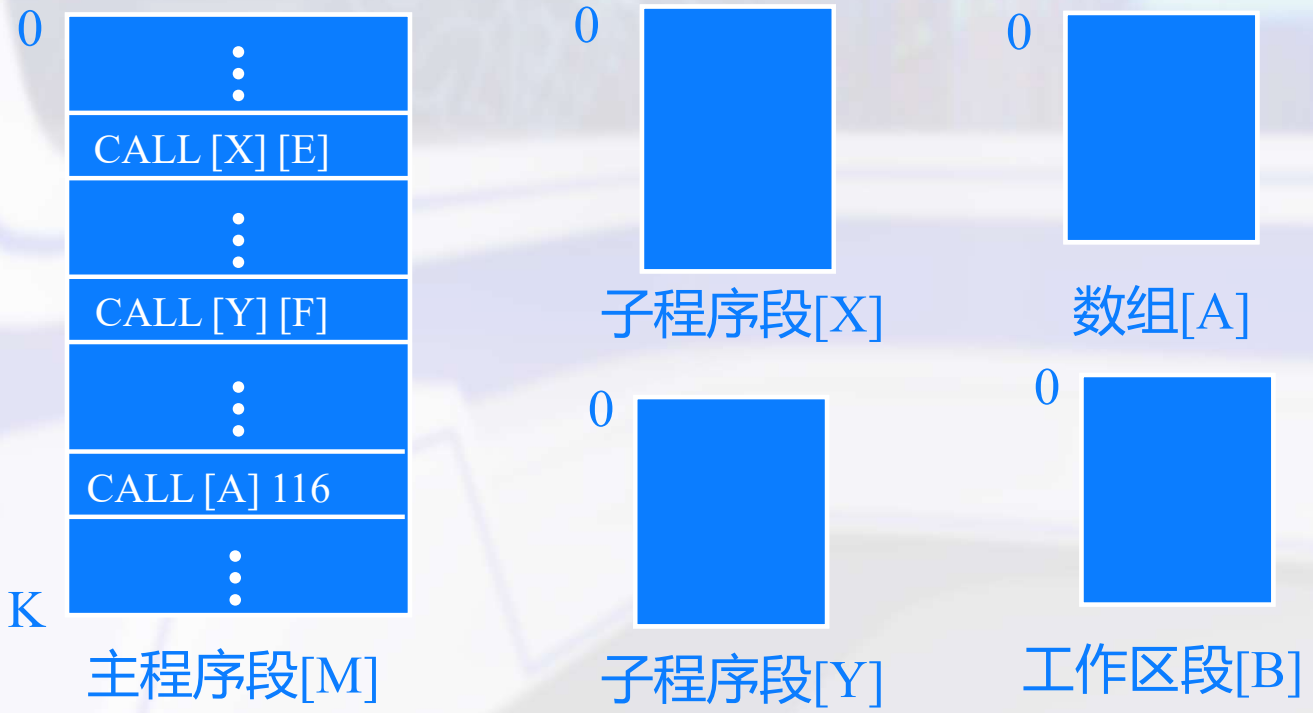
1 逻辑分段

- 按程序逻辑关系划分为若干程序段或数据段
- 段从0开始编号，每一段内部从0开始编址

段号	段内地址

逻辑分段与内存划分

1 逻辑分段



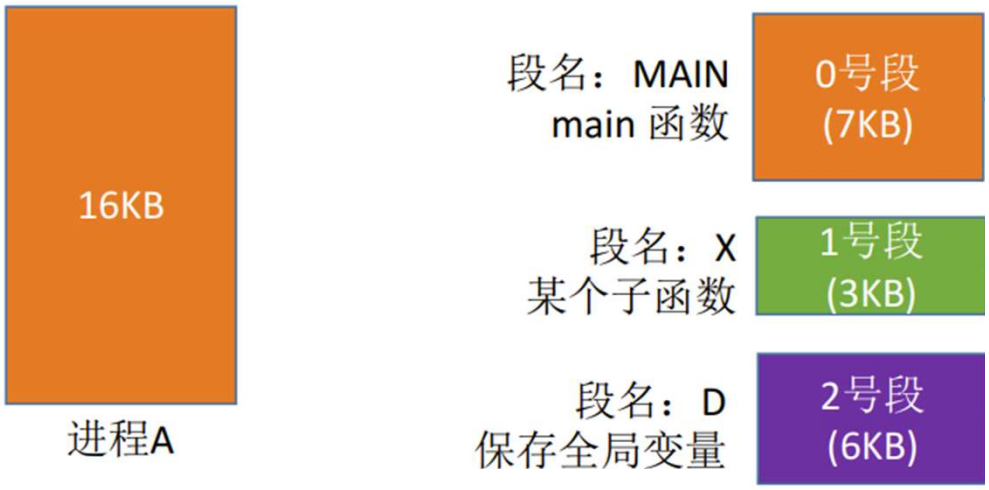
逻辑分段与内存划分

2 内存划分

- 内存空间按需划分为若干个长度不相同的物理段
- 每个物理段由起始地址和长度确定
- 每段在内存中占据连续空间，段间可以不连续存放

分段

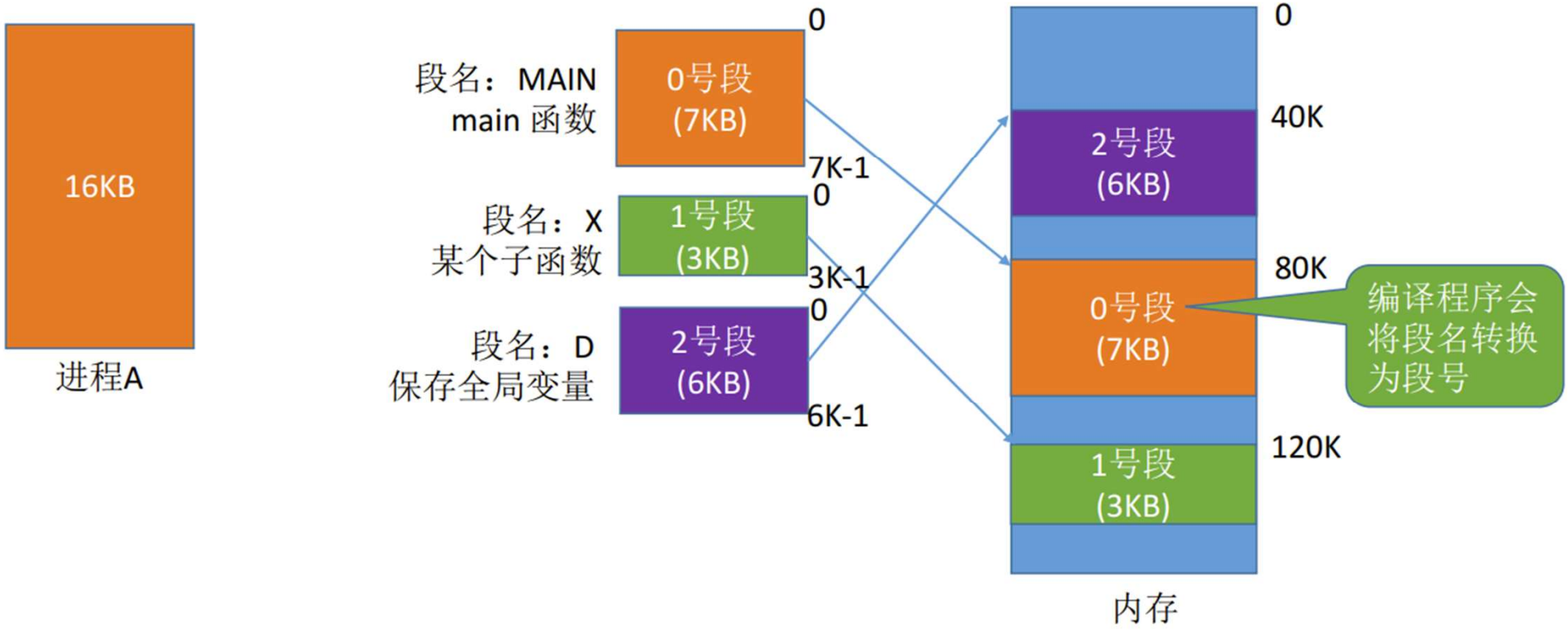
进程的地址空间：按照程序自身的逻辑关系划分为若干个段，每个段都有一个段名（在低级语言中，程序员使用段名来编程），每段从0开始编址



分段

进程的地址空间：按照程序自身的逻辑关系划分为若干个段，每个段都有一个段名（在低级语言中，程序员使用段名来编程），每段从0开始编址

内存分配规则：以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻。

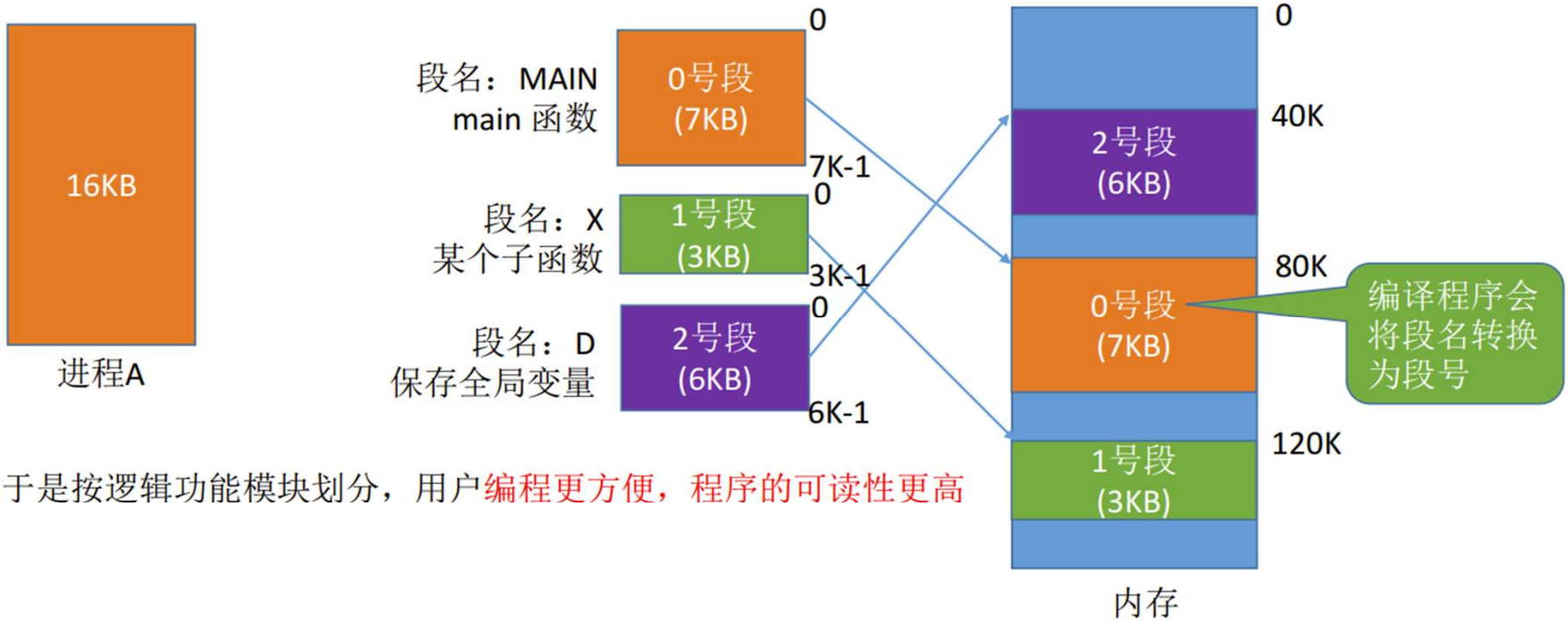


分段

按需分配大小，不再是固定大小

进程的地址空间：按照程序自身的逻辑关系划分为若干个段，每个段都有一个段名（在低级语言中，程序员使用段名来编程），每段从0开始编址

内存分配规则：以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻。



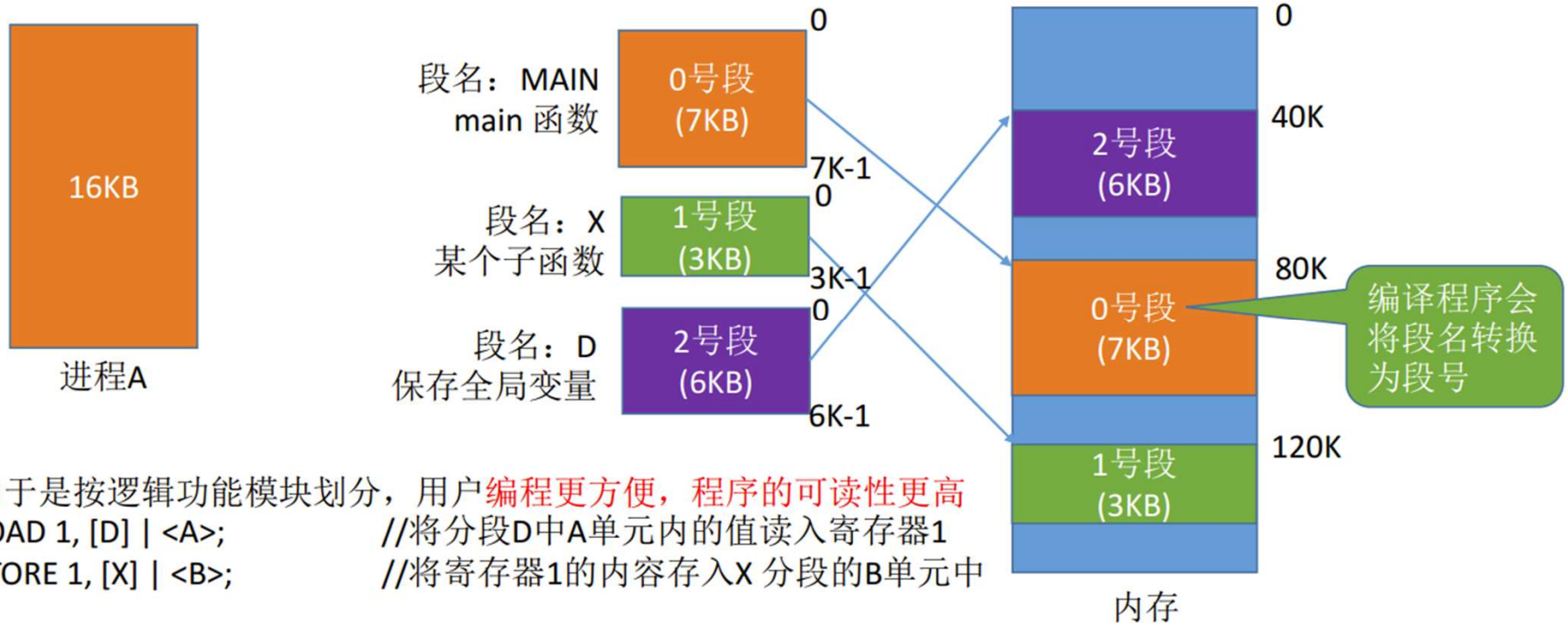
由于是按逻辑功能模块划分，用户编程更方便，程序的可读性更高

分段

按需分配大小，不再是固定大小

进程的地址空间：按照程序自身的逻辑关系划分为若干个段，每个段都有一个段名（在低级语言中，程序员使用段名来编程），每段从0开始编址

内存分配规则：以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻。



由于是按逻辑功能模块划分，用户编程更方便，程序的可读性更高

```
LOAD 1, [D] | <A>;          //将分段D中A单元内的值读入寄存器1
STORE 1, [X] | <B>;         //将寄存器1的内容存入X 分段的B单元中
```

分段

分段系统的逻辑地址结构由段号（段名）和段内地址（段内偏移量）所组成。如：

31 …… 16	15 …… 0
段号	段内地址

段号的位数决定了每个进程最多可以分几个段
段内地址位数决定了每个段的最大长度是多少

分段

分段系统的逻辑地址结构由段号（段名）和段内地址（段内偏移量）所组成。如：

31 …… 16	15 …… 0
段号	段内地址

段号的位数决定了每个进程最多可以分几个段
段内地址位数决定了每个段的最大长度是多少

在上述例子中，若系统是按字节寻址的，则
段号占16位，因此在该系统中，每个进程最多有 2^{16} = 64K 个段
段内地址占 16位，因此每个段的最大长度是 2^{16} = 64KB。

分段

分段系统的逻辑地址结构由段号（段名）和段内地址（段内偏移量）所组成。如：

31 …… 16	15 …… 0
段号	段内地址

段号的位数决定了每个进程最多可以分几个段
段内地址位数决定了每个段的最大长度是多少

在上述例子中，若系统是按字节寻址的，则
段号占16位，因此在该系统中，每个进程最多有 2^{16} = 64K 个段
段内地址占 16位，因此每个段的最大长度是 2^{16} = 64KB。

```
LOAD 1, [D] | <A>;          //将分段D中A单元内的值读入寄存器1
STORE 1, [X] | <B>;         //将寄存器1的内容存入X 分段的B单元中
```

写程序时使用的段名 [D]、[X] 会被编译程序翻译成对应段号

<A>单元、<B>单元会被编译程序翻译成段内地址

分段

分段系统的逻辑地址结构由段号（段名）和段内地址（段内偏移量）所组成。如：

31 …… 16	15 …… 0
段号	段内地址

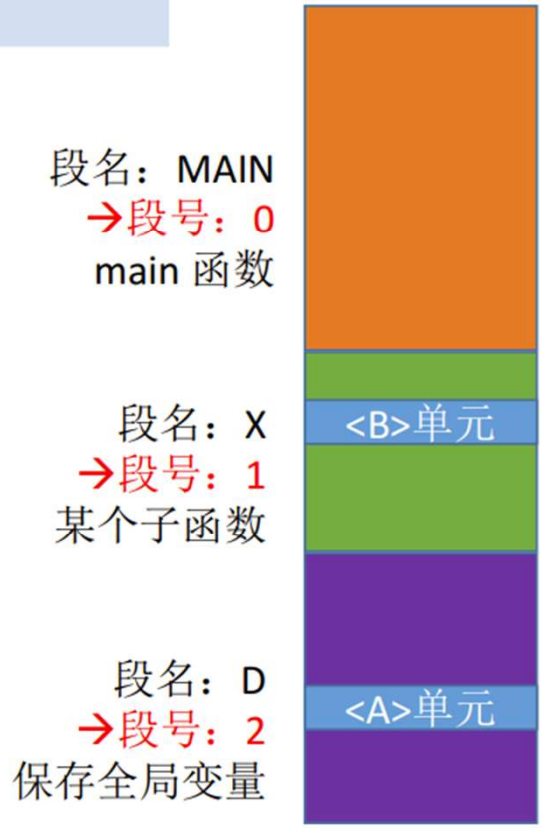
段号的位数决定了每个进程最多可以分几个段
段内地址位数决定了每个段的最大长度是多少

在上述例子中，若系统是按字节寻址的，则
段号占16位，因此在该系统中，每个进程最多有 2^{16} = 64K 个段
段内地址占 16位，因此每个段的最大长度是 2^{16} = 64KB。

```
LOAD 1, [D] | <A>;        //将分段D中A单元内的值读入寄存器1
STORE 1, [X] | <B>;       //将寄存器1的内容存入X 分段的B单元中
```

写程序时使用的段名 [D]、[X] 会被编译程序翻译成对应段号

<A>单元、<B>单元会被编译程序翻译成段内地址



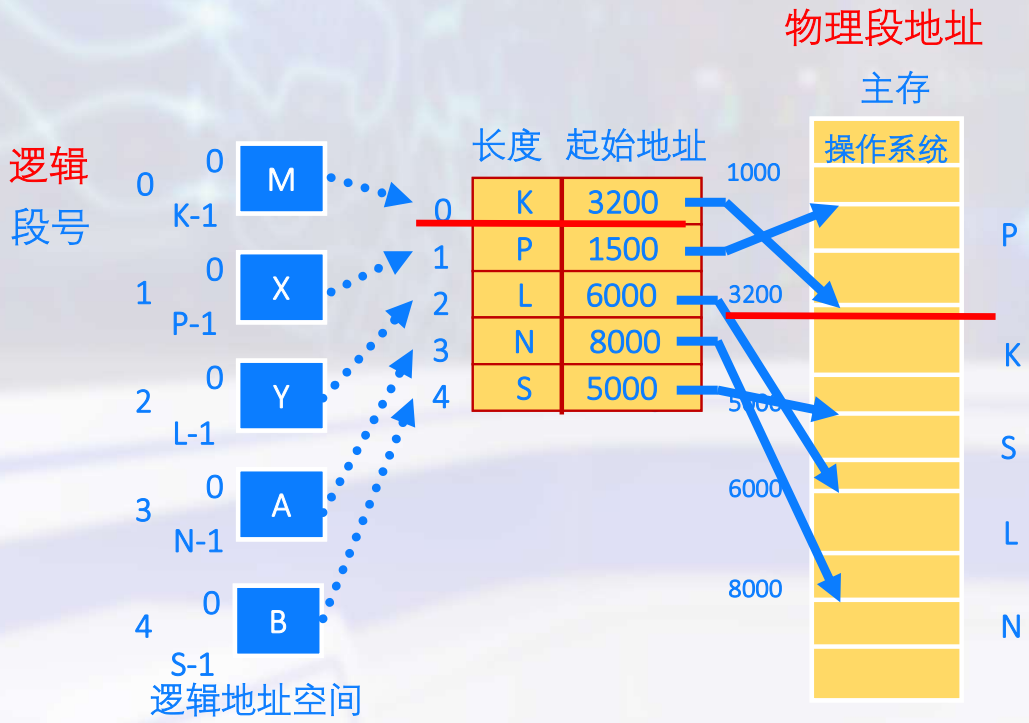
分段存储管理机制

本讲内容

1. 逻辑分段与内存划分
2. 分段存储的管理表格
3. 分段存储的地址转换
4. 分页和分段存储比较
5. 段页式存储管理方案

分段存储的管理表格

1 段表



页表：*逻辑页面*到实际的*物理页框*的映射关系
每一段都是固定长度不需要记录

段表：*逻辑段*到实际的*物理内存地址*的映射关系
每段长度都不一样，需要记录长度

分段存储的管理表格

1 段表

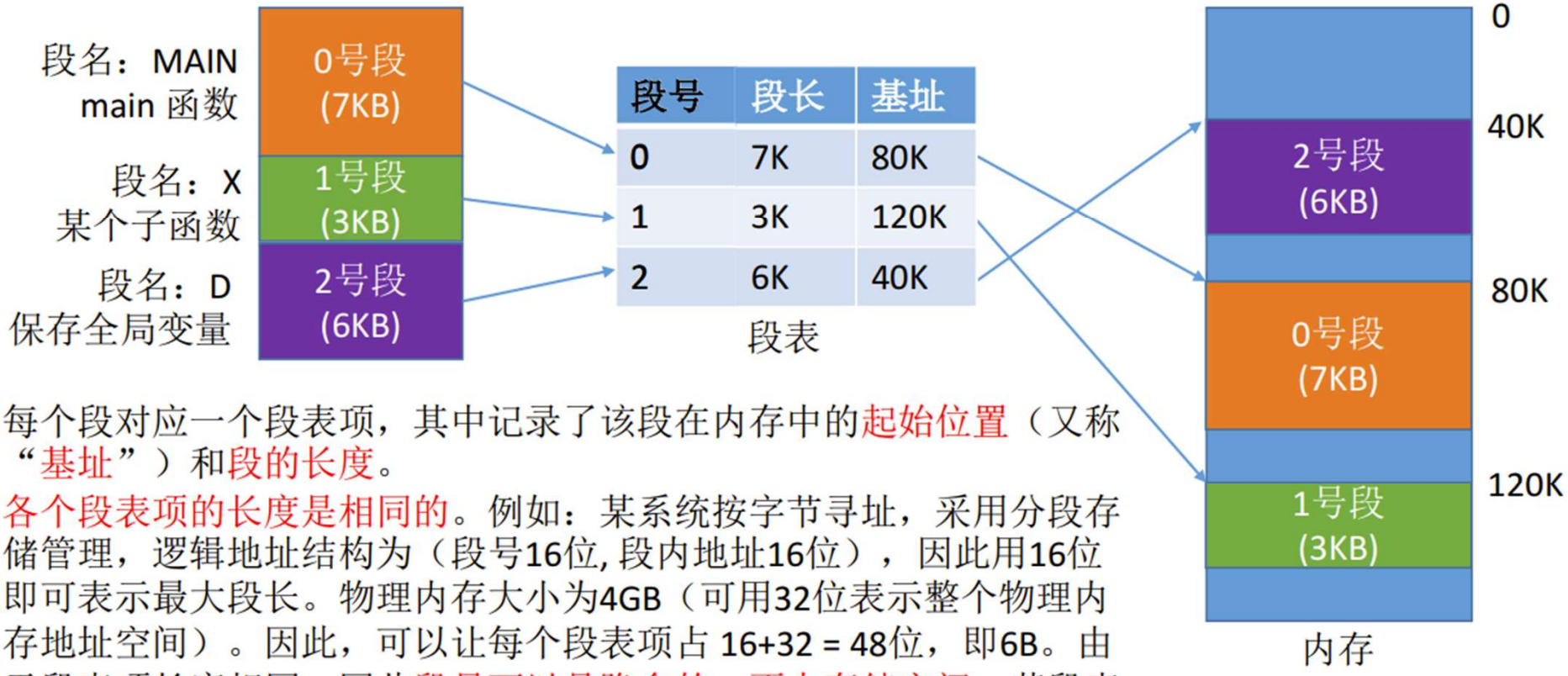
- 每个进程一张段表，给出逻辑段和物理段对应关系
- 段表放在内存，属于进程的现场信息
- 段表在进程装入内存时，根据内存分配情况建立

逻辑段号	起始地址	长度
0	58	20
1	100	110
2	260	140

重点记录这三个属性：
逻辑段号、起始地址、长度

段表

问题：程序分多个段，各段离散地装入内存，为了保证程序能正常运行，就必须能从物理内存中找到各个逻辑段的存放位置。为此，需为每个进程建立一张段映射表，简称“段表”。



1. 每个段对应一个段表项，其中记录了该段在内存中的起始位置（又称“基址”）和段的长度。
2. 各个段表项的长度是相同的。例如：某系统按字节寻址，采用分段存储管理，逻辑地址结构为（段号16位, 段内地址16位），因此用16位即可表示最大段长。物理内存大小为4GB（可用32位表示整个物理内存地址空间）。因此，可以让每个段表项占 16+32 = 48位，即6B。由于段表项长度相同，因此段号可以是隐含的，不占存储空间。若段表存放的起始地址为 M，则 K号段对应的段表项存放的地址为 M + K*6

分段存储的管理表格

2 硬件支持

- 段表始址寄存器：

  用于保存正在运行进程的段表的始址

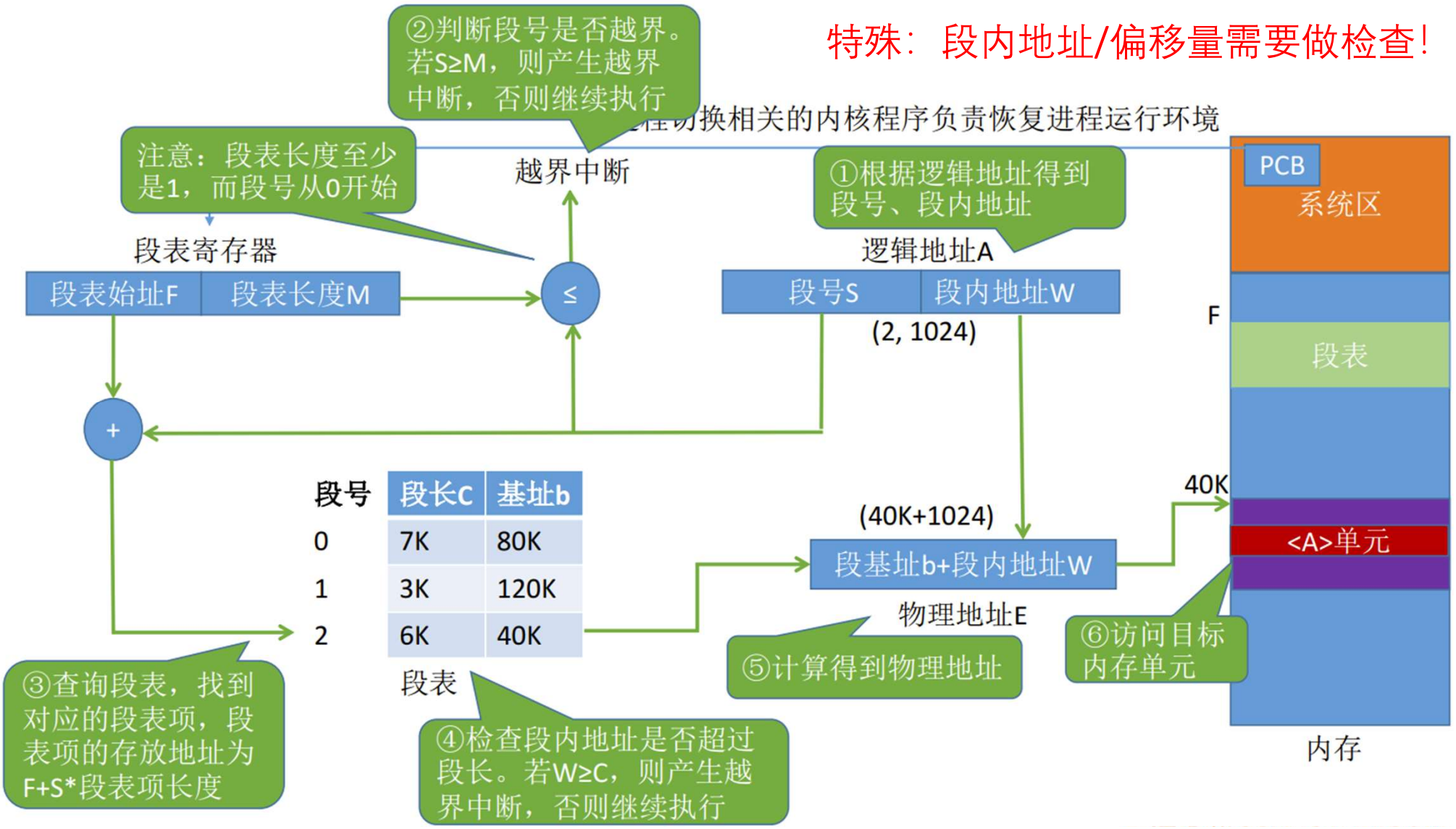
- 段表长度寄存器：

  用于保存正在运行进程的段表的长度

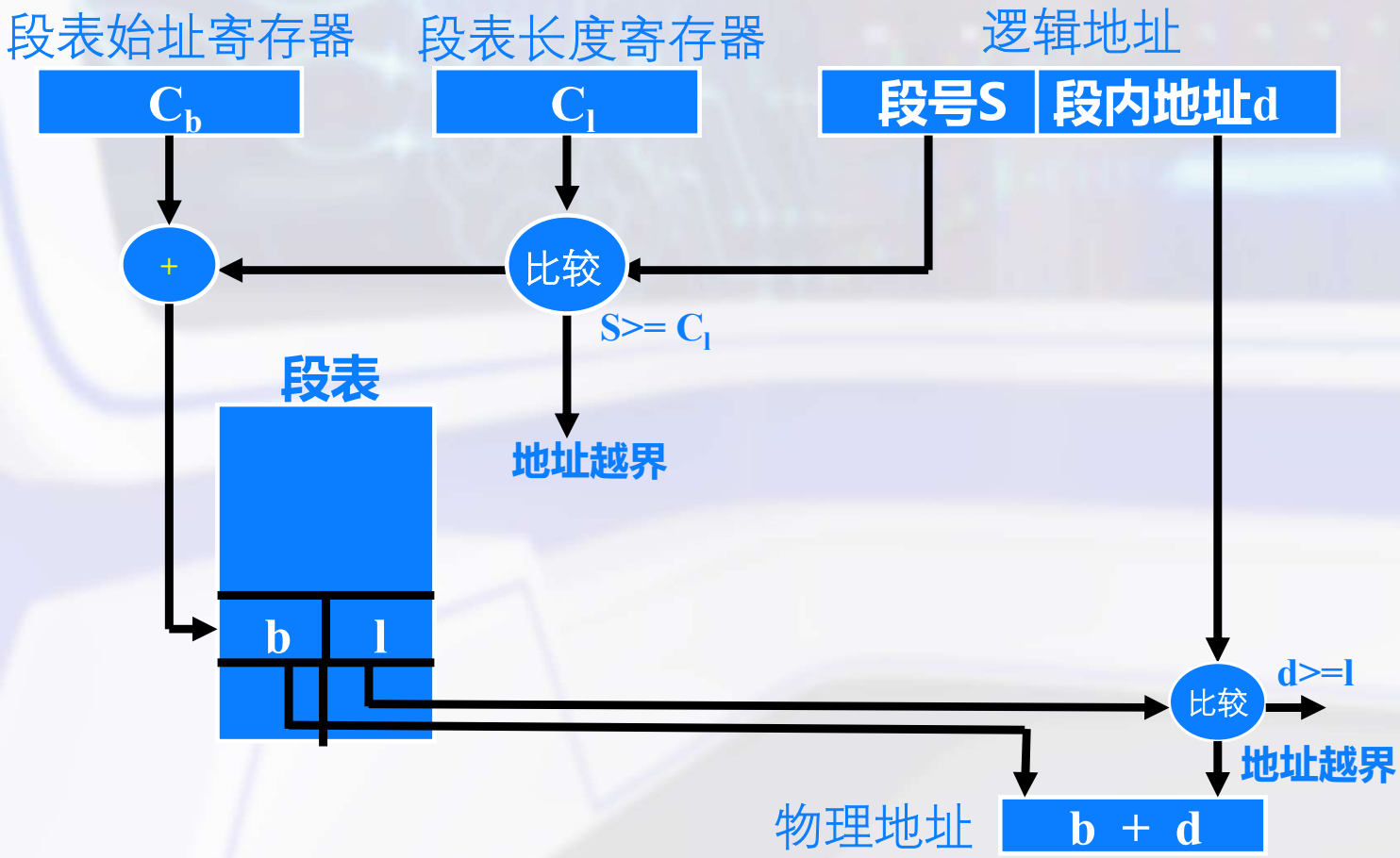
分段存储管理机制

本讲内容

1. 逻辑分段与内存划分
2. 分段存储的管理表格
3. 分段存储的地址转换
4. 分页和分段存储比较
5. 段页式存储管理方案


特殊：段内地址/偏移量需要做检查！！
②判断段号是否越界。若S≥M，则产生越界中断，否则继续执行
程切换相关的内核程序负责恢复进程运行环境
注意：段表长度至少是1，而段号从0开始
越界中断
①根据逻辑地址得到段号、段内地址
段表寄存器
逻辑地址A
段表始址F
段表长度M
≤
段号S
段内地址W
(2, 1024)
PCB
系统区
F
段表
+
段号
段长C
基址b
0
7K
80K
1
3K
120K
2
6K
40K
(40K+1024)
40K
段基址b+段内地址W
<A>单元
物理地址E
⑤计算得到物理地址
⑥访问目标内存单元
③查询段表，找到对应的段表项，段表项的存放地址为F+S*段表项长度
段表
④检查段内地址是否超过段长。若W≥C，则产生越界中断，否则继续执行
内存

分段存储的地址转换


段表始址寄存器
段表长度寄存器
逻辑地址
C_b
C_l
段号S
段内地址d
+
比较
S>= C_l
段表
地址越界
b
l
比较
d>=l
地址越界
物理地址
b + d

分段存储管理机制

本讲内容

1. 逻辑分段与内存划分
2. 分段存储的管理表格
3. 分段存储的地址转换
4. 分页和分段存储比较
5. 段页式存储管理方案

分页和分段存储比较

1 分页存储特点

优点

- 解决了碎片问题
- 便于管理

缺点

- 不易实现共享
- 不便于动态连接

分页和分段存储比较

2 分段存储特点

优点

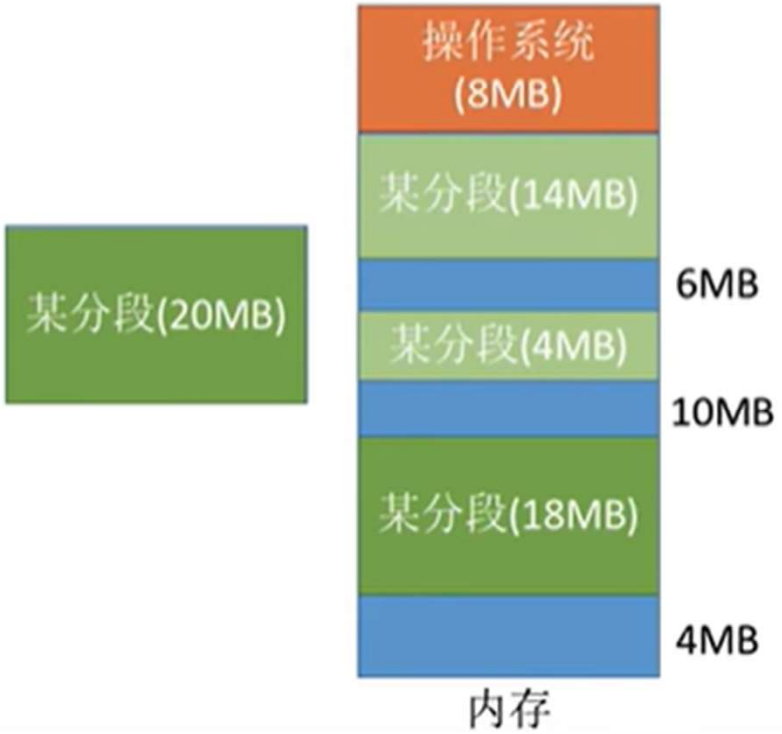
- 便于动态申请内存
- 段表长度较短
- 便于共享
- 便于动态链接

缺点

- 产生碎片
- 不易扩展

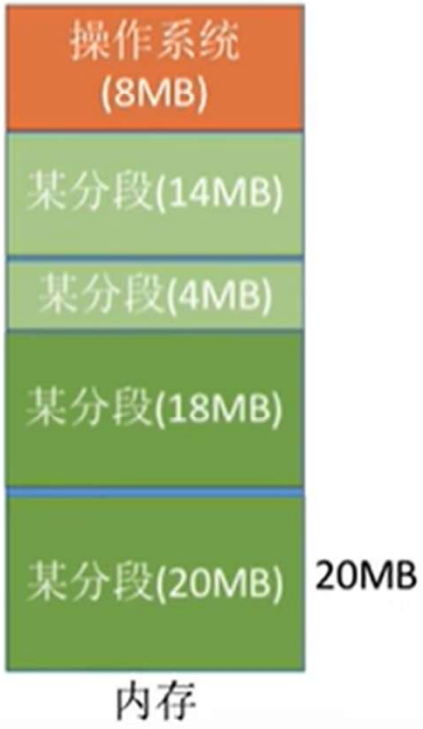
分页、分段的优缺点分析

	优点	缺点
分页管理	内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理会产生外部碎片



分页、分段的优缺点分析

	优点	缺点
分页管理	内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理会产生外部碎片



分段管理中产生的外部碎片也可以用“紧凑”来解决，只是需要付出较大的时间代价

分段、分页管理的对比

页是信息的物理单位。分页的主要目的是为了实现离散分配，提高内存利用率。分页仅仅是系统管理上的需要，完全是系统行为，对用户是不可见的。

段是信息的逻辑单位。分段的主要目的是更好地满足用户需求。一个段通常包含着一组属于一个逻辑模块的信息。分段对用户是可见的，用户编程时需要显式地给出段名。

页的大小固定且由系统决定。段的长度却不固定，决定于用户编写的程序。

分页的用户进程地址空间是一维的，程序员只需给出一个记忆符即可表示一个地址。
分段的用户进程地址空间是二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。

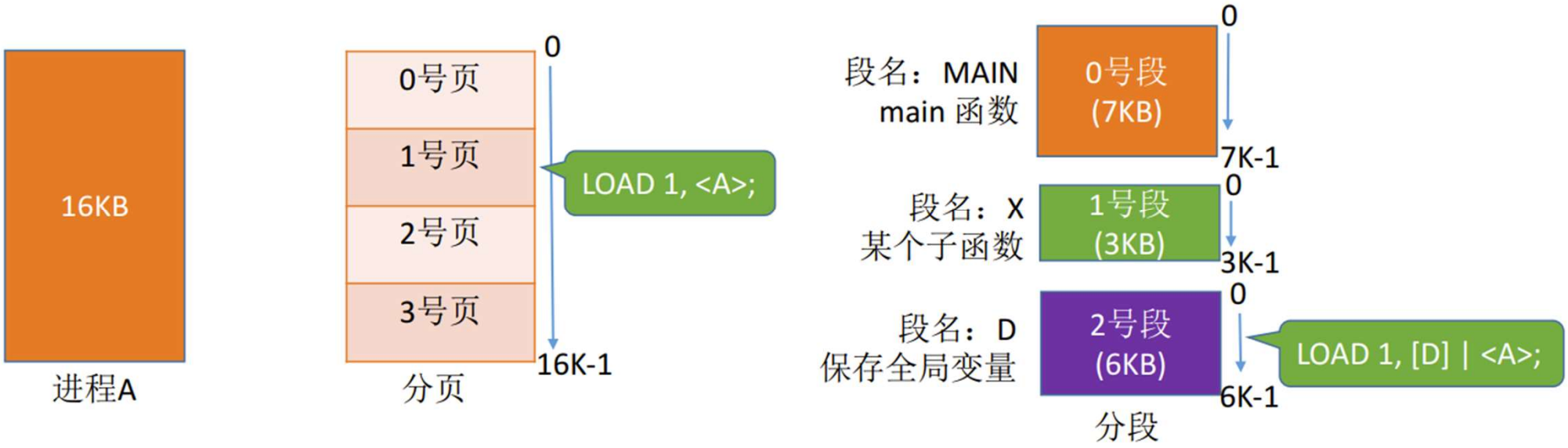
分段、分页管理的对比

页是信息的物理单位。分页的主要目的是为了实现离散分配，提高内存利用率。分页仅仅是系统管理上的需要，完全是系统行为，对用户是不可见的。

段是信息的逻辑单位。分段的主要目的是更好地满足用户需求。一个段通常包含着一组属于一个逻辑模块的信息。分段对用户是可见的，用户编程时需要显式地给出段名。

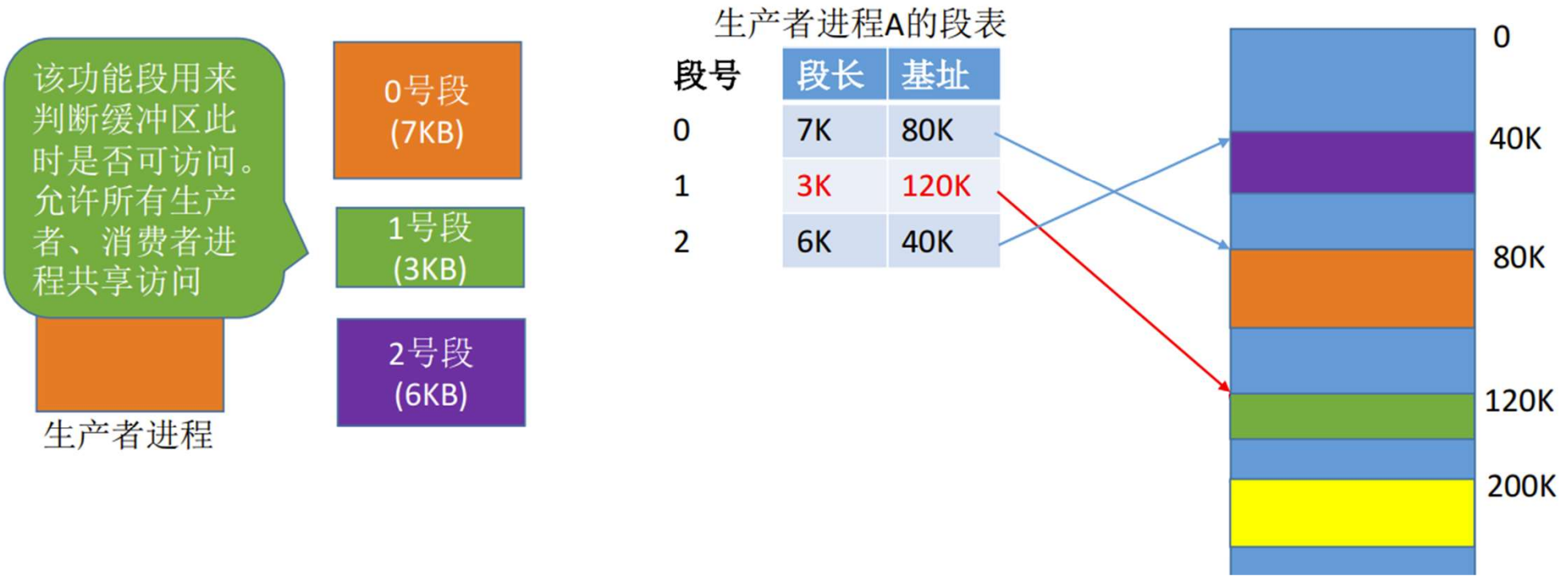
页的大小固定且由系统决定。段的长度却不固定，决定于用户编写的程序。

分页的用户进程地址空间是一维的，程序员只需给出一个记忆符即可表示一个地址。
分段的用户进程地址空间是二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。



分段、分页管理的对比

分段比分页更容易实现信息的共享和保护。



分段、分页管理的对比

比如，有一个代码段只是简单的输出“Hello World！”

分段比分页更容易实现信息的共享和保护。

不能被修改的代码称为纯代码或可重入代码（不属于临界资源），这样的代码是可以共享的。可修改的代码是不能共享的（比如，有一个代码段中有很多变量，各进程并发地同时访问可能造成数据不一致）

该功能段用来判断缓冲区此时是否可访问。允许所有生产者、消费者进程共享访问

生产者进程

将生产者进程分段

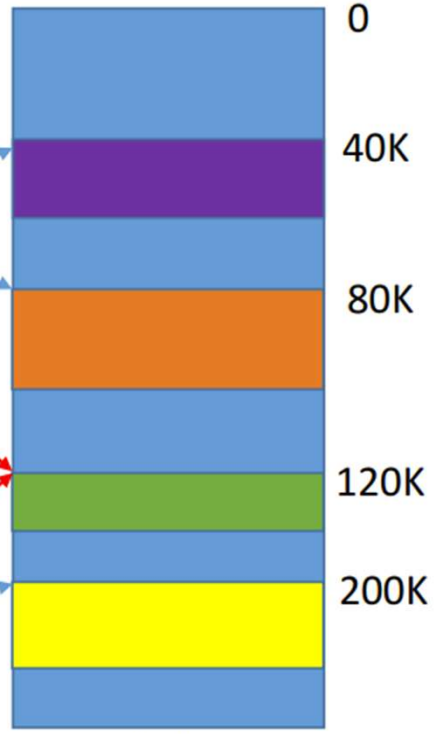
生产者进程A的段表

段号	段长	基址
0	7K	80K
1	3K	120K
2	6K	40K

消费者进程B的段表

段号	段长	基址
0	21K	200K
1	3K	120K

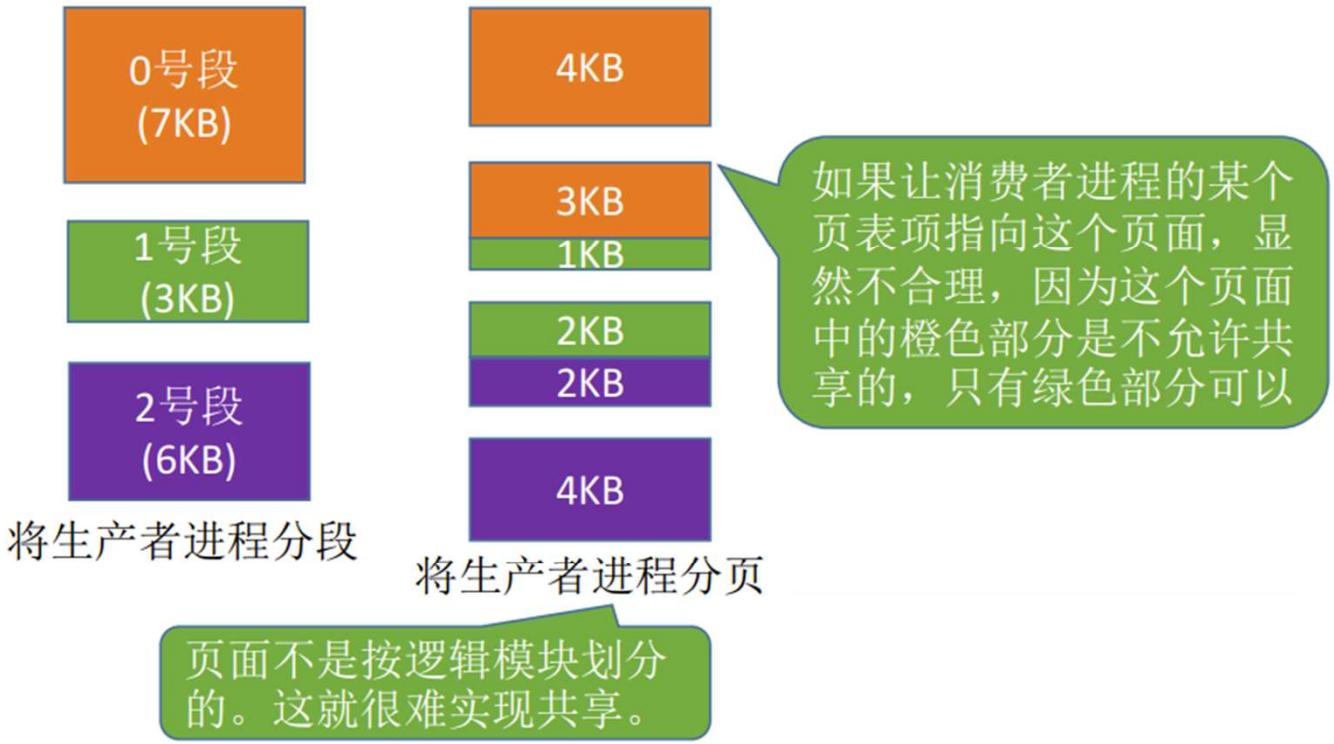
只需让各进程的段表项指向同一个段即可实现共享



内存

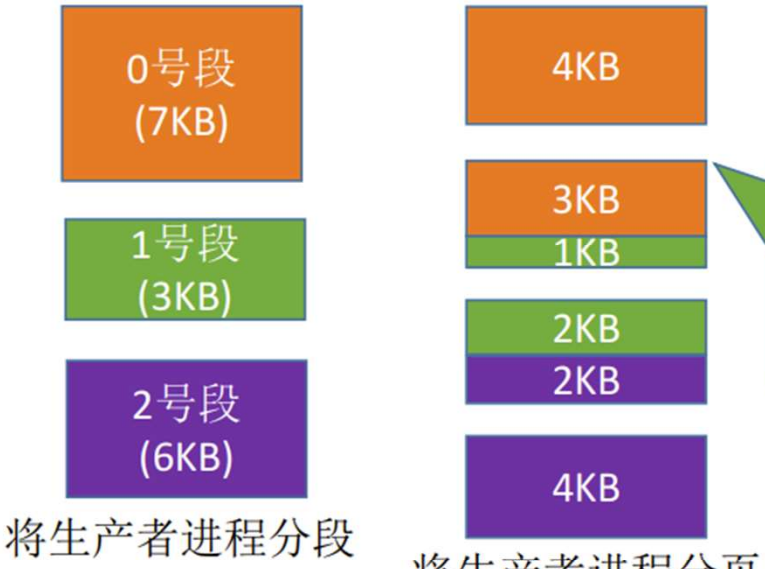
分段、分页管理的对比

分段比分页更容易实现信息的共享和保护。



分段、分页管理的对比

分段比分页更容易实现信息的共享和保护。



如果让消费者进程的某个页表项指向这个页面，显然不合理，因为这个页面中的橙色部分是不允许共享的，只有绿色部分可以

页面不是按逻辑模块划分的。这就很难实现共享。

1、2号页面只有一部分允许其他进程访问，因此很难用页表实现信息保护

生产者进程A的段表

段号	段长	基址	是否允许其他进程访问
0	7K	80K	不允许
1	3K	120K	允许
2	6K	40K	不允许

生产者进程A的页表

页号	基址	是否允许其他进程访问
0	…	不允许
1	…	~~允许~~
2	…	~~允许~~
3	…	不允许

分段存储管理机制

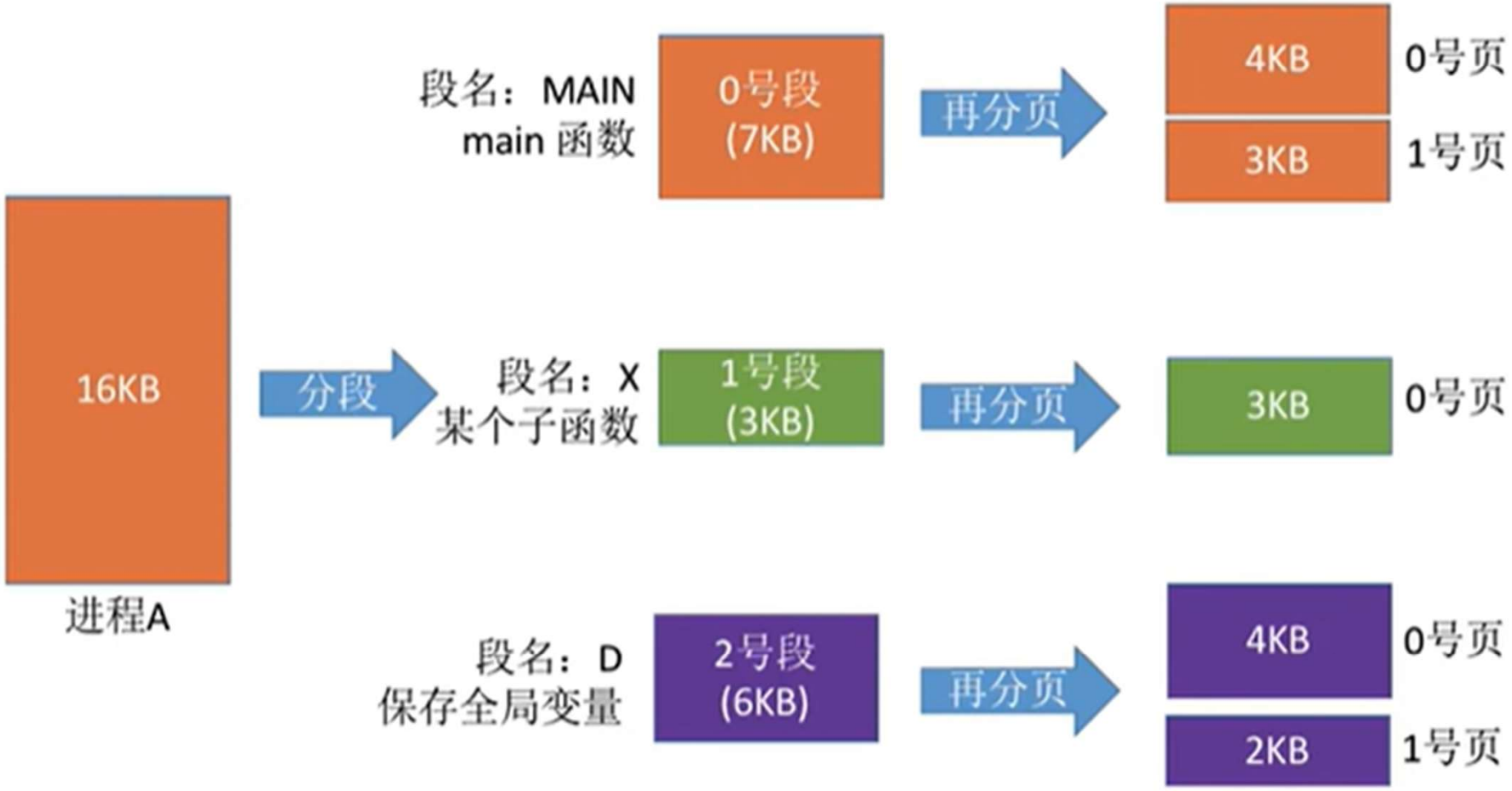
本讲内容

1. 逻辑分段与内存划分
2. 分段存储的管理表格
3. 分段存储的地址转换
4. 分页和分段存储比较
5. 段页式存储管理方案

段页式存储管理方案

- 结合页式段式优点，克服二者的缺点
- 按逻辑关系划分用户程序为若干逻辑段
- 按分页存储管理机制划分和分配内存

分段+分页=段页式管理



将进程按逻辑模块分段，再将各段分页（如每个页面4KB）
再将内存空间分为大小相同的内存块/页框/页帧/物理块

段页式管理的逻辑地址结构

分段系统的逻辑地址结构由段号和段内地址（段内偏移量）组成。如：

31 …… 16	15 …… 0
段号	段内地址

段页式系统的逻辑地址结构由段号、页号、页内地址（页内偏移量）组成。如：

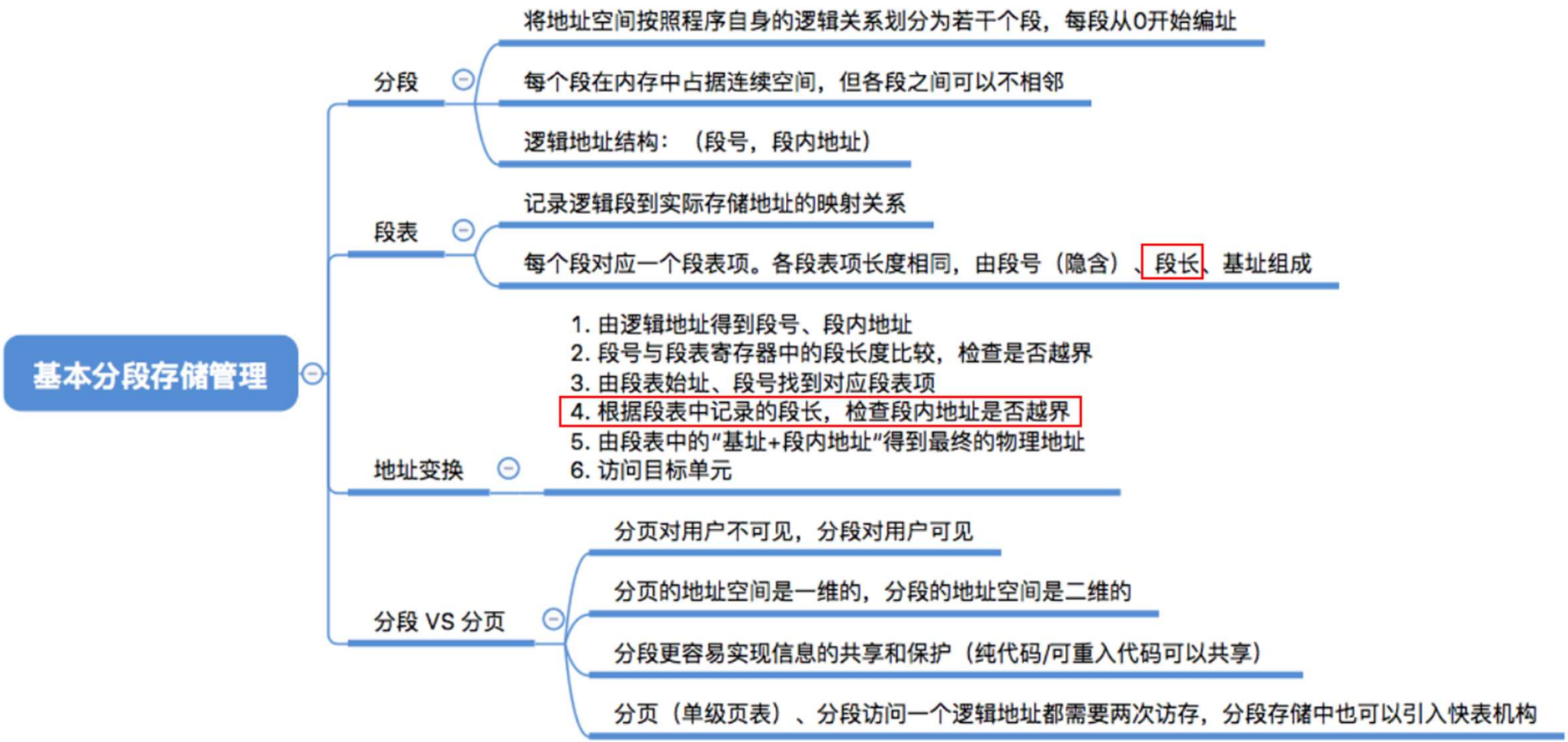
31 …… 16	15 …… 12	11 …… 0
段号	页号	页内偏移量

段号的位数决定了每个进程最多可以分几个段
页号位数决定了每个段最大有多少页
页内偏移量决定了页面大小、内存块大小是多少

在上述例子中，若系统是按字节寻址的，则
段号占16位，因此在该系统中，每个进程最多有 $2^{16} = 64K$ 个段
页号占4位，因此每个段最多有 $2^4 = 16$ 页
页内偏移量占12位，因此每个页面\每个内存块大小为 $2^{12} = 4096 = 4KB$

“分段”对用户是可见的，程序员编程时需要显式地给出段号、段内地址。而将各段“分页”对用户是不可见的。系统会根据段内地址自动划分页号和页内偏移量。
因此段页式管理的地址结构是二维的。

知识回顾与重要考点


基本分段存储管理
分段
将地址空间按照程序自身的逻辑关系划分为若干个段，每段从0开始编址
每个段在内存中占据连续空间，但各段之间可以不相邻
逻辑地址结构：（段号，段内地址）
段表
记录逻辑段到实际存储地址的映射关系
每个段对应一个段表项。各段表项长度相同，由段号（隐含）、段长、基址组成
地址变换
1. 由逻辑地址得到段号、段内地址
2. 段号与段表寄存器中的段长度比较，检查是否越界
3. 由段表始址、段号找到对应段表项
4. 根据段表中记录的段长，检查段内地址是否越界
5. 由段表中的“基址+段内地址”得到最终的物理地址
6. 访问目标单元
分段 VS 分页
分页对用户不可见，分段对用户可见
分页的地址空间是一维的，分段的地址空间是二维的
分段更容易实现信息的共享和保护（纯代码/可重入代码可以共享）
分页（单级页表）、分段访问一个逻辑地址都需要两次访存，分段存储中也可以引入快表机构