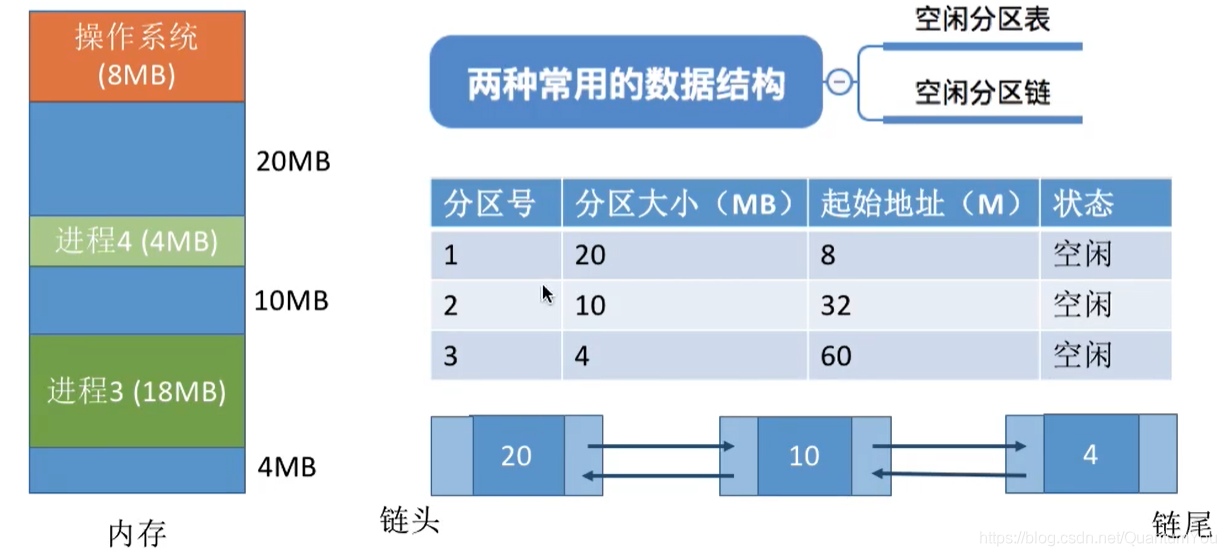
前程似锦，未来可期

@[toc]

# 动态分区分配算法

* 动态分区分配算法：在动态分区分配方式中，当很多个空闲分区都能满足需求时，应该选择哪个分区进行分配？

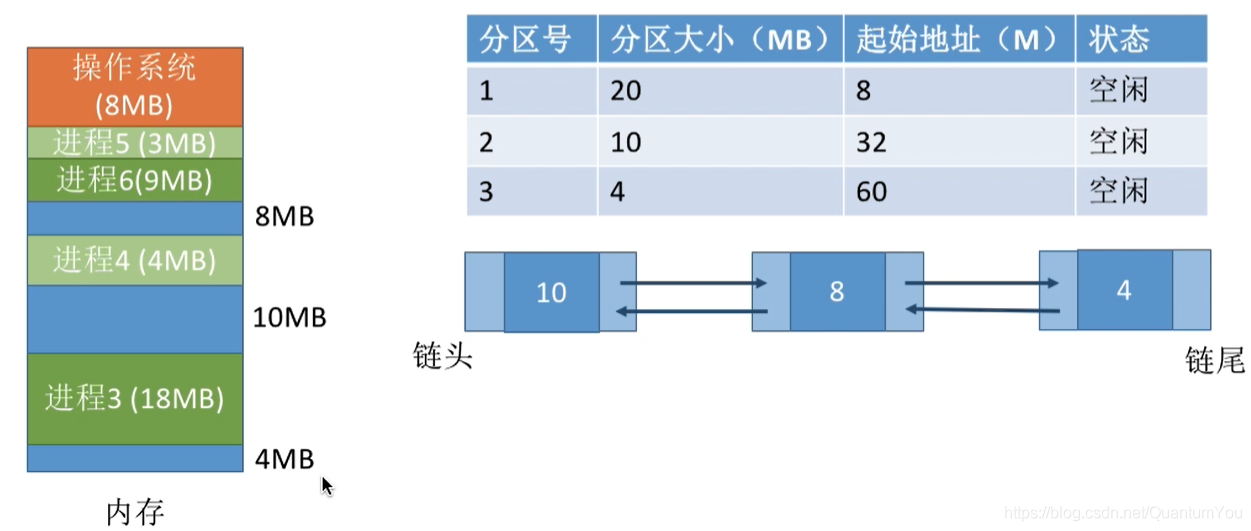
## 首次适应算法（First Fit）

* **算法思想**：每次都从低地址开始査找，找到第一个能满足大小的空闲分区。
* **如何实现**：空闲分区以地址递增的次序排列。每次分配内存时顺序查找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。  
  

## 最佳适应算法（Best Fit）

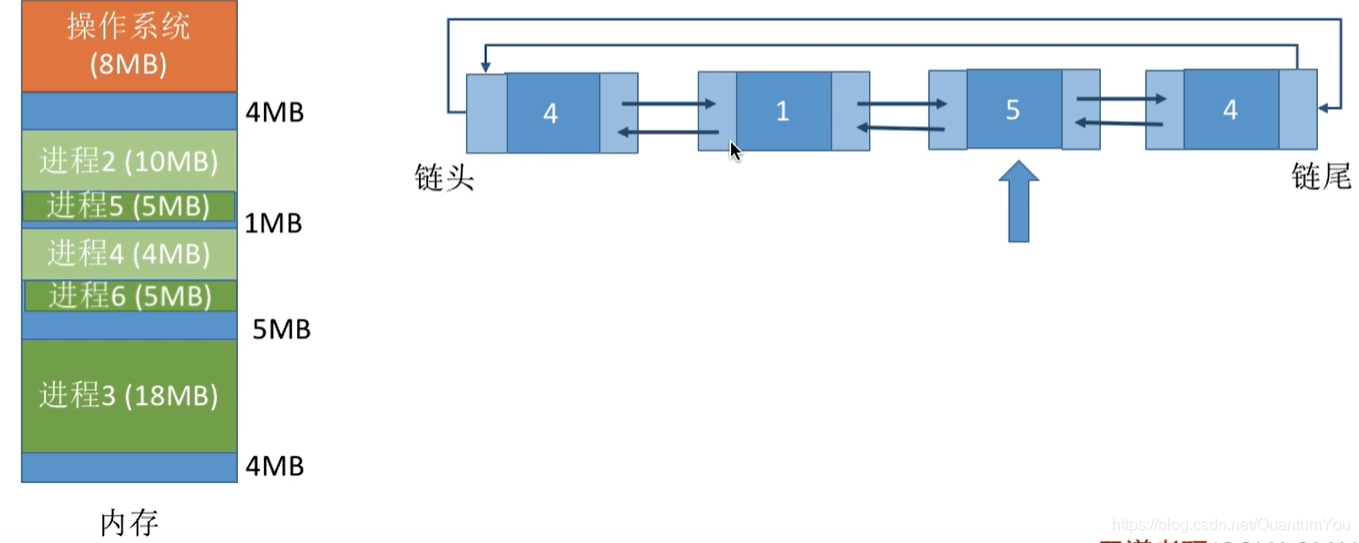
* **算法思想**：由于动态分区分配是一种连续分配方式，为各进程分配的空间必须是连续的一整片区域。因此为了保证当“大进程”到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区，即，优先使用更小的空闲区。
* **如何实现**：空闲分区按容量递增次序链接。每次分配内存时顺序査找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。
* 缺点：每次都选最小的分区进行分配，会留下越来越多的、很小的、难以利用的内存块。因此这种方法会产生很多的外部碎片。

## 最坏适应算法（Worst Fit）

* 又称最大适应算法（ Largest Fit）
* **算法思想**：为了解决最佳适应算法的问题一一即留下太多难以利用的小碎片，可以在每次分配时优先使用最大的连续空闲区，这样分配后剩余的空闲区就不会太小，更方便使用。
* **如何实现**：空闲分区按容量递减次序链接。每次分配内存时顺序查找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。
* **缺点**：每次都选最大的分区进行分配，虽然可以让分配后留下的空闲区更大，更可用，但是这种方式会导致较大的连续空闲区被迅速用完。如果之后有“大进程”到达，就没有内存分区可用了。  
  

## 领近适应算法（Next Fit）

* **算法思想**：首次适应算法每次都从链头开始査找的。这可能会导致低地址部分出现很多小的空闲分区，而每次分配査找时，都要经过这些分区，因此也增加了査找的开销。如果每次都从上次查找结束的位置开始检索，就能解决上述问题。
* **如何实现**：空闲分区以地址递增的顺序排列（可排成一个循环链表）.每次分配内存时从上次查找结束的位置开始査找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。



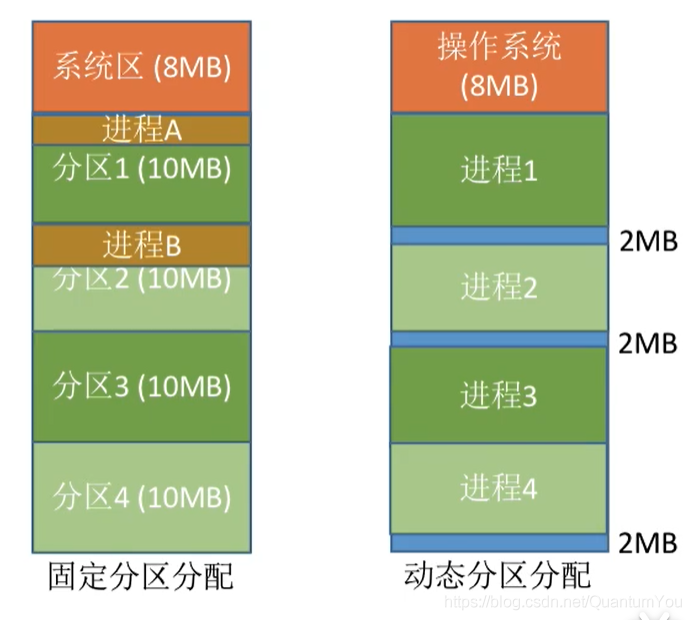
* **分析**： 首次适应算法每次都要从头査找，每次都需要检索低地址的小分区。但是这种规则也决定了当低地址部分有更小的分区可以满足需求时，会更有可能用到低地址部分的小分区，也会更有可能把高地址部分的大分区保留下来（最佳适应算法的优点）
* 邻近适应算法的规则可能会导致无论低地址、高地址部分的空闲分区都有相同的概率被使用，也就导致了高地址部分的大分区更可能被使用，划分为小分区，最后导致无大分区可用（最大适应算法的缺点）
* 这样导致反而首次适应算法效果更好

## 小结



# 基本分页存储管理

## 连续分配方式的缺点

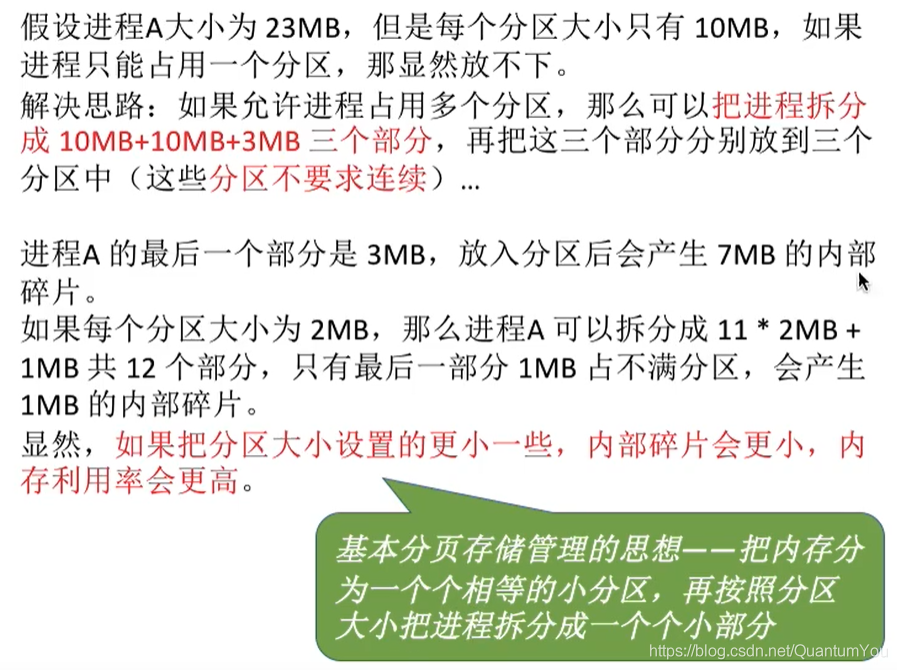
* 考虑支持多道程序的两种连续分配方式  
  1、**固定分区分配**：缺乏灵活性，会产生大量的内部碎片，内存的利用率很低。  
  2、**动态分区分配**：会产生很多外部碎片，虽然可以用“紧凑”技术来处理，但是“紧凑”的时间代价很高  
  

改进： 如果允许将一个进程分散地装入到许多不相邻的分区中，便可充分地利用内存，而无需再进行“紧凑”------------>基于这一思想，产生了“非连续分配方式”，或者称为“离散分配方式”

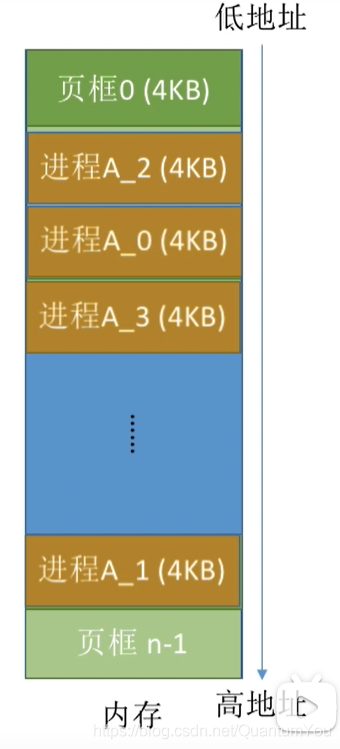
* 连续分配：为用户进程分配的必须是一个连续的内存空间。
* 非连续分配：为用户进程分配的可以是一些分散的内存空间。

## 内部碎片外部碎片

* 动态分区分配没有内部碎片，但是有外部碎片。
* **内部碎片**：分配给某进程的内存区域中，如果有些部分没有用上
* **外部碎片**：是指内存中的某些空闲分区由于太小而难以利用



## 分页存储管理的基本概念

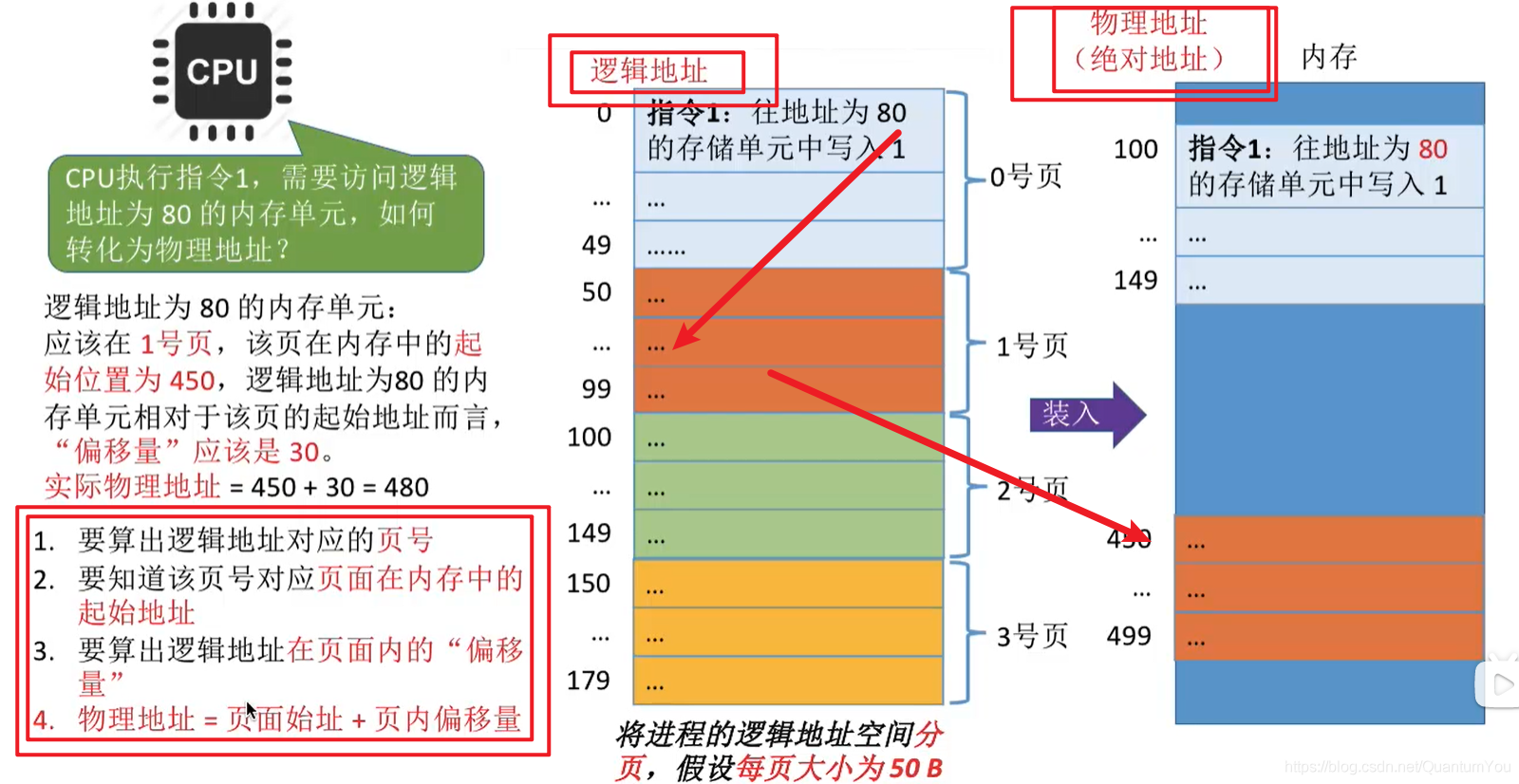
* 将内存空间分为一个个大小相等的分区（比如：每个分区4KB），每个分区就是一个“页框”，或称“页帧”、“内存块”、“物理块”.每个页框有一个编号，即“页框号”（或者“内存块号”、“页帧号”、“物理块号”）页框号**从0开始**。
* 将用户进程的地址空间也分为与页框大小相等的一个个区域称为“页”或“页面”.每个页面也有一个编号，即“页号”页号也是**从0开始**。（注：进程的最后一个页面卤能没有一个页框那么大。因此，页框不能太大，否则可能产生过大的内部碎片）
* 操作系统**以页框为单位为各个进程分配**内存空间。进程的每个页面分别放入一个页框中。也就是说，进程的页面与内存的页框有一一对应的关系。各个页面不必连续存放，也不必按先后顺序来，可以放到不相邻的各个页框中。  
  

## 实现地址的转换

* 将进程地址空间分页之后，操作系统该如何实现逻辑地址到物理地址的转换？

[跳转链接](https://blog.csdn.net/QuantumYou/article/details/118637618?spm=1001.2014.3001.5501)  


* **动态重定位**：进程在内存中连续存放，操作系统实现从逻辑地址到物理地址的转换
* **思想**：模块在内存中的起始地址（重定位寄存器）+目标内存单元相对于起始地址的偏移量



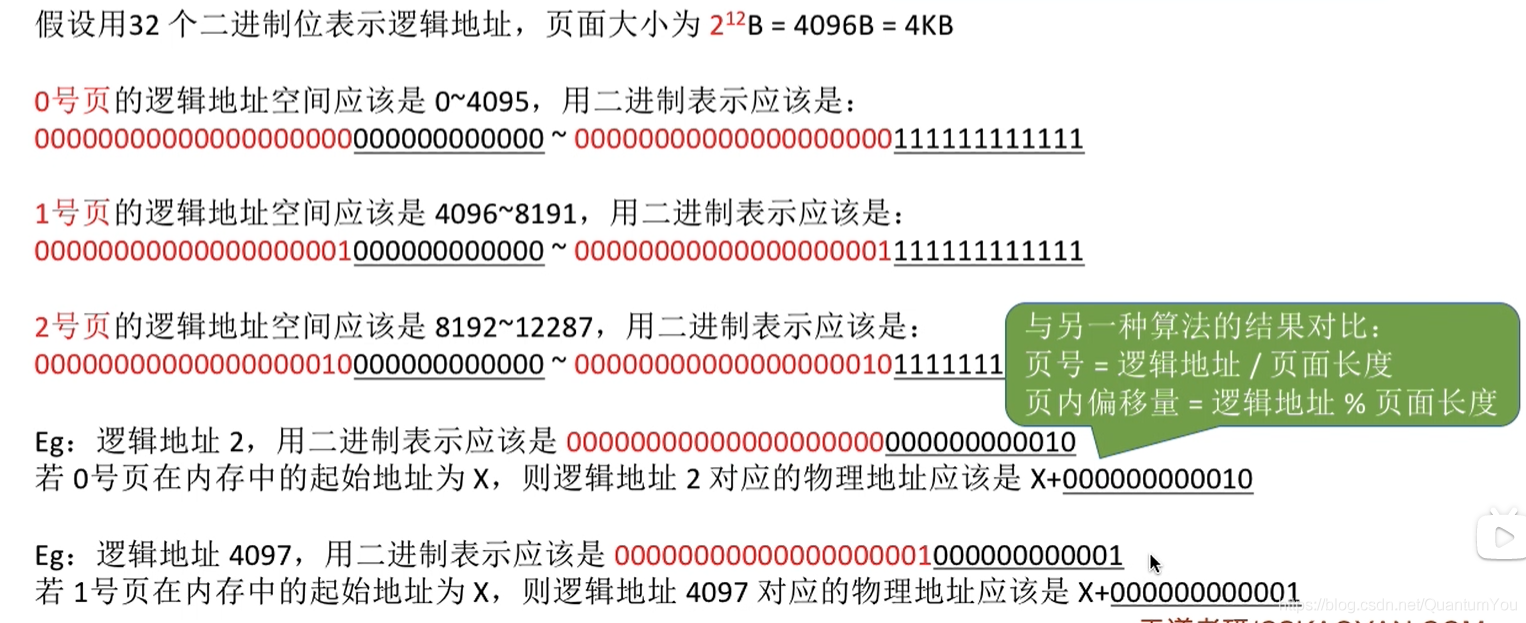
### 计算方法：

人工计算方法：

* 页号=逻辑地址/页面长度（取除法的整数部分）
* 页内偏移量≡逻辑地址%页面长度（取除法的余数部分）
* 页面在内存中的起始位置：操作系统需要用某种数据结构记录进程各个页面的起始位置。

Eg:  
页号=80/50=1   
页内偏移量=80%50=30   
1号页在内存中存放的起始位置450

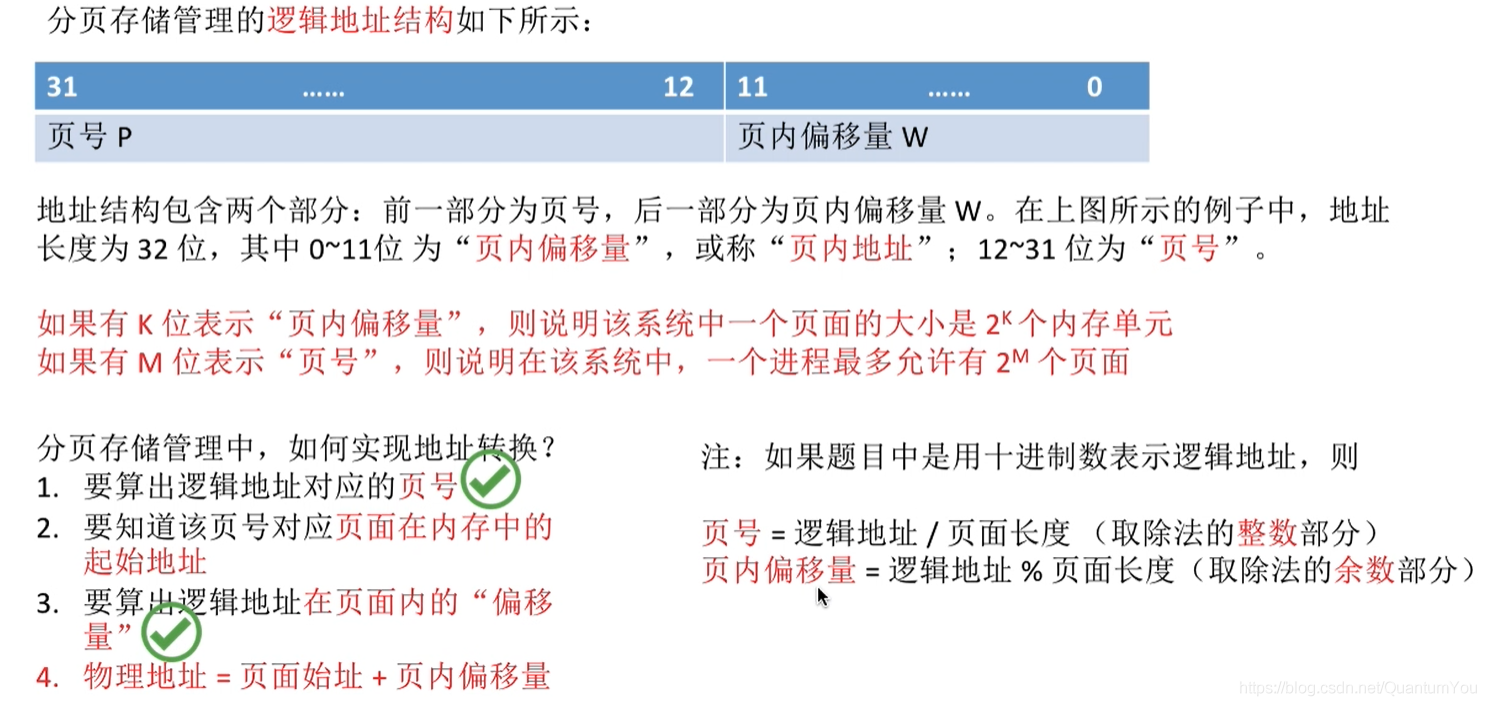
计算机计算方法

* 后面12位表示页内偏移量  
  

知识交叉：

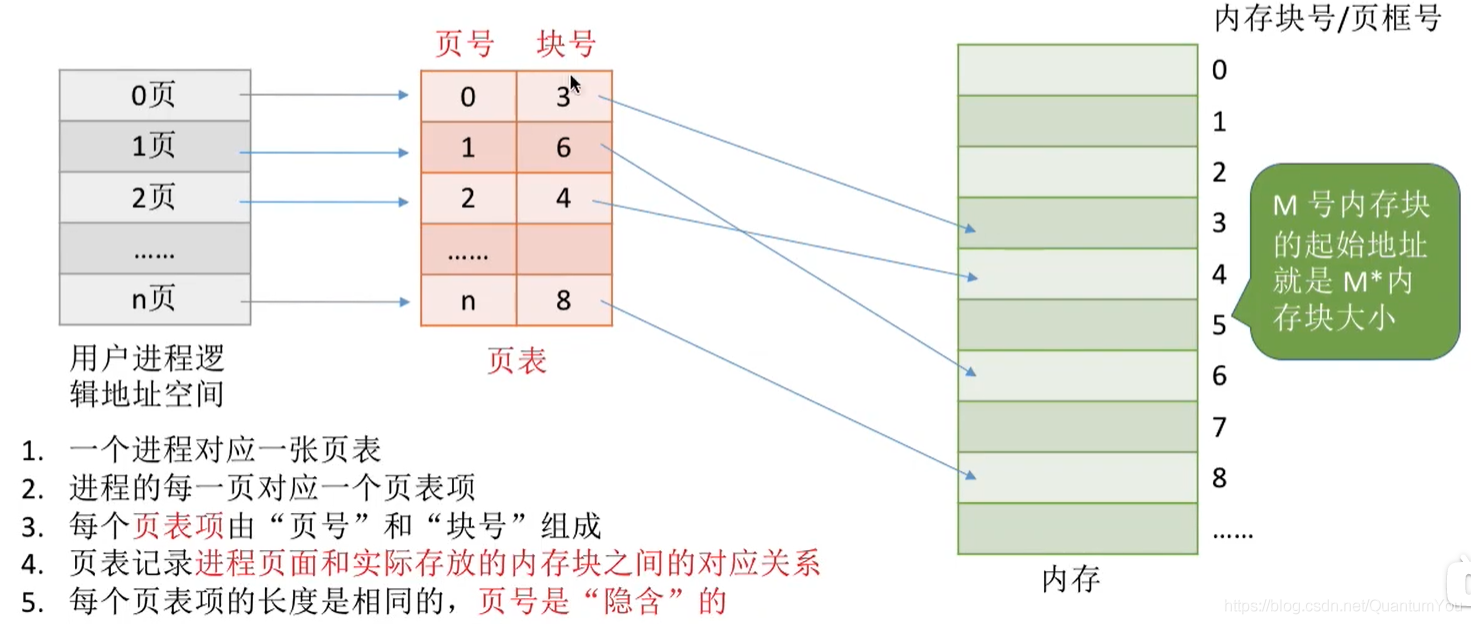
* cache 块内地址总是2的整数幂
* 全相联映射、组相联映射、直接映射
* 子网划分的掩码
* **重要结论**：如果每个页面大小为2KB，用二进制数表示逻辑地址，则末尾K位即为页内偏移量，其余部分就是页号

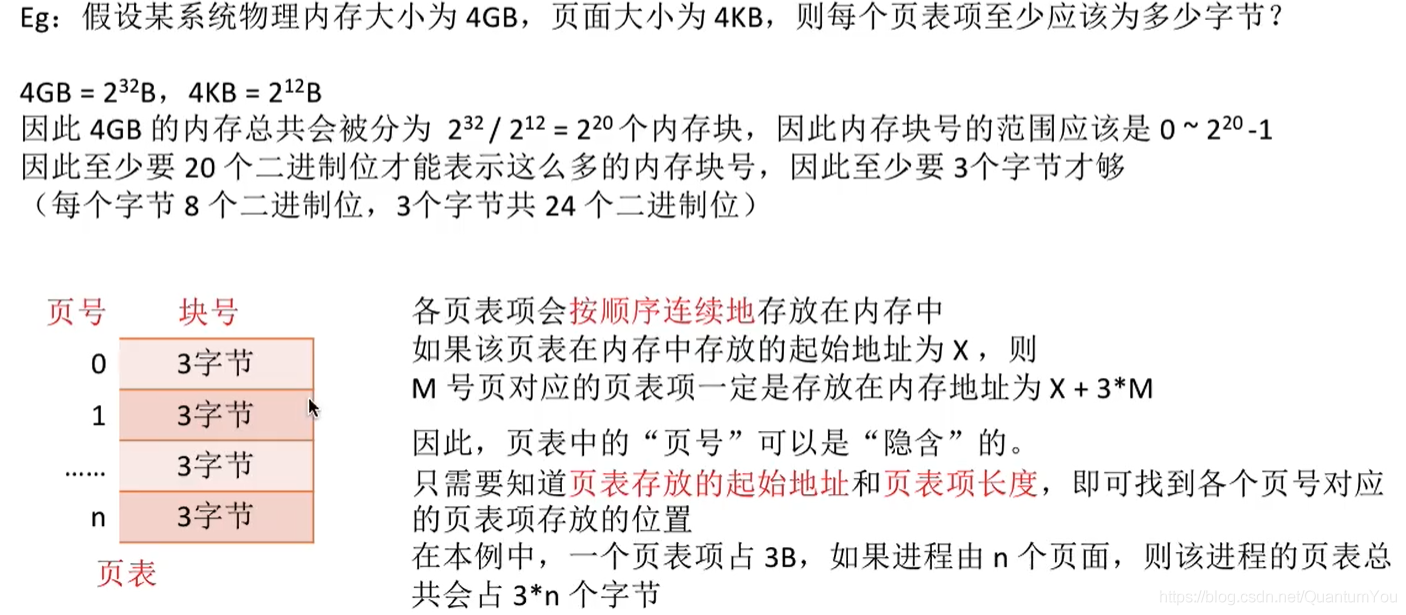
### 逻辑地址结构



### 页表

* 为了能知道进程的每个页面在内存中存放的位置，操作系统要为每个进程建立一张页表



* 为什么每个**页表项的长度是相同**的，页号是“隐含”的？  
  

## 小结

