

现代操作系统 (Modern Operating System)

福州大学 计数学院 江兰帆



第3章 存储管理

- 内存（RAM）是计算机中一种需要认真管理的重要资源。就目前来说，虽然一台普通家用计算机的内存容量已经是20世纪60年代早期全球最大的计算机IBM 7094的内存容量的10 000倍以上，但是程序大小的增长速度比内存容量的增长速度要快得多。在这一章中，我们将讨论操作系统是怎样对内存创建抽象模型以及怎样管理内存的。

内存的作用

- 内存是由存储单元（字节或字）组成的一维连续地址空间，用来存放当前正在运行的程序的代码或数据。内存负责向L1、L2缓存提供内容。

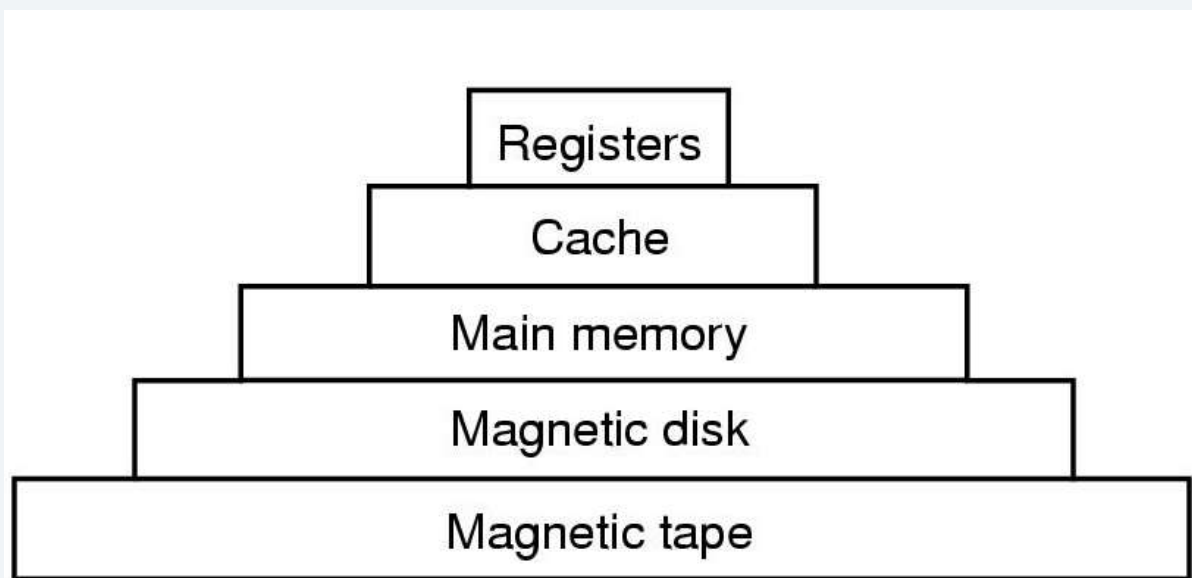
对于内存的要求

- 内存的直接存取速度尽量快到与CPU取指速度相匹配，其容量大到能装下当前运行的程序与数据，否则CPU执行速度就会受到内存速度和容量的影响而得不到充分发挥。

存储器的层次结构

- 在现代计算机系统中，存储器是信息的来源与归宿，占据重要位置。但是，在现有技术条件下，任何一种存储装置，都无法同时从速度与容量两方面，满足用户的需求。实际上它们组成了一个速度由快到慢，容量由小到大的存储装置层次。

计算机的存储体系回顾



各种速度和容量的存储器硬件，在操作系统协调之下形成了一种存储器层次结构，或称存储体系。

多级cache结构



为什么需要存储管理？

- 帕金森定律：有多大内存，就有多大程序。
- 存储器一直以来都是较为珍贵的系统资源，需要合理使用。
- 程序的逻辑空间和实际的物理空间不甚相同，需要进行映射。

存储管理的任务

- 主存的分配和管理：当用户需要内存时,系统为之分配相应的存储空间；不需要时，及时回收，以供其它用户使用。
- 提高主存储器的利用率：不仅能使多道程序动态地共享主存，提高主存利用率，最好还能共享主存中某个区域的信息。

存储管理的任务

- “扩充”主存容量：为用户提供比主存物理空间大得多的地址空间，以至使用户感觉他的作业是在这样一个大的存储器中运行。
- 存储保护：确保多道程序都在各自分配到的存储区域内操作，互不干扰，防止一道程序破坏其它作业或系统文件的信息。

基本概念

- 物理地址
- 逻辑地址
- 符号地址/名地址

内存的物理组织

- 把内存分为若干个大
小相等的存储单元，
每个单元给一个编号，
这个编号称为内存地
址(物理地址、实地
址)，存储单元占8位，
称为字节(Byte)。

	7	6	5	4	3	2	1	0 bit
N-1	0	1	0	0	0	1	1	0
N-2	0	0	1	0	1	1	0	1
.	0	1	1	1	0	0	1	0
.	1	0	0	1	1	0	0	1
.	1	1	1	1	0	0	1	0
2	0	0	1	0	1	0	0	1
1	1	0	1	1	0	0	0	1
0	0	0	0	1	0	1	1	0

物理地址空间

- 物理地址的集合称为物理地址空间(主存地址空间)，它是一个一维的线性地址空间。

逻辑空间

- 源程序经过汇编或编译后，形成目标程序，每个目标程序都是以0为基址顺序进行编址的，原来用符号名访问的单元用具体的数据——单元号取代。这样生成的目标程序占据一定的地址空间，称为作业的逻辑地址空间，简称逻辑空间。在逻辑空间中每条指令的地址和指令中要访问的操作数地址统称为逻辑地址。

符号地址、逻辑地址、物理地址

程序在成为进程前的准备工作：

- 编辑：形成源文件(符号地址)
- 编译：形成目标模块
- 链接：多个目标模块或程序库生成可执行文件（逻辑地址）
- 装入：构造PCB，形成进程(物理地址)

作业的名空间、逻辑地址空间和装入后的物理空间

名空间

Mov R1,[data]

data: 6817

逻辑空间

0:

50: Mov R1,[200]

200: 6817

物理空间

1000:

1050: Mov R1,[200]

1200: 6817

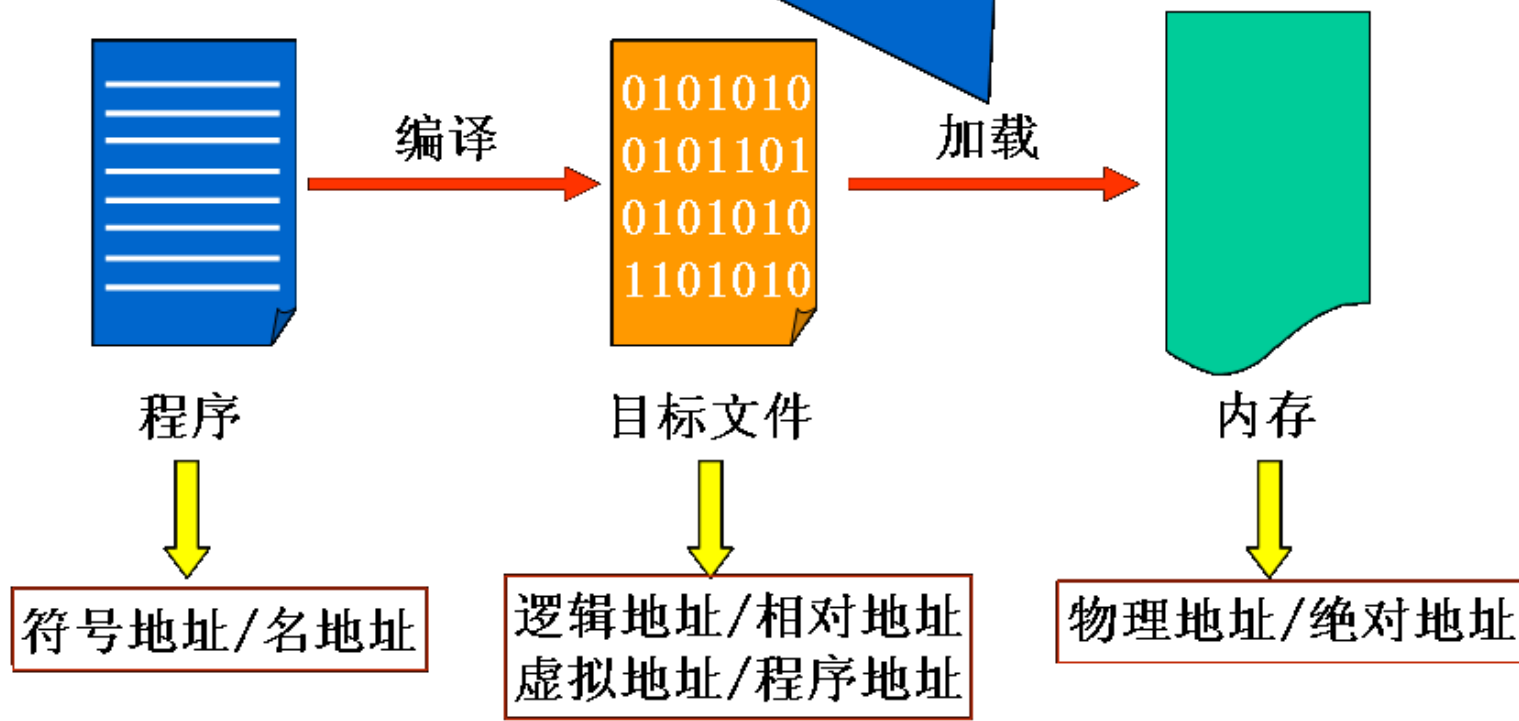
问题的提出

- 逻辑地址→物理地址？ ？ ？

地址重定位

基本概念

定义：当程序被装入内存时，程序的逻辑地址被转换成内存的物理地址，这一过程称为地址重定位（由内存管理单元(MMU)完成）。



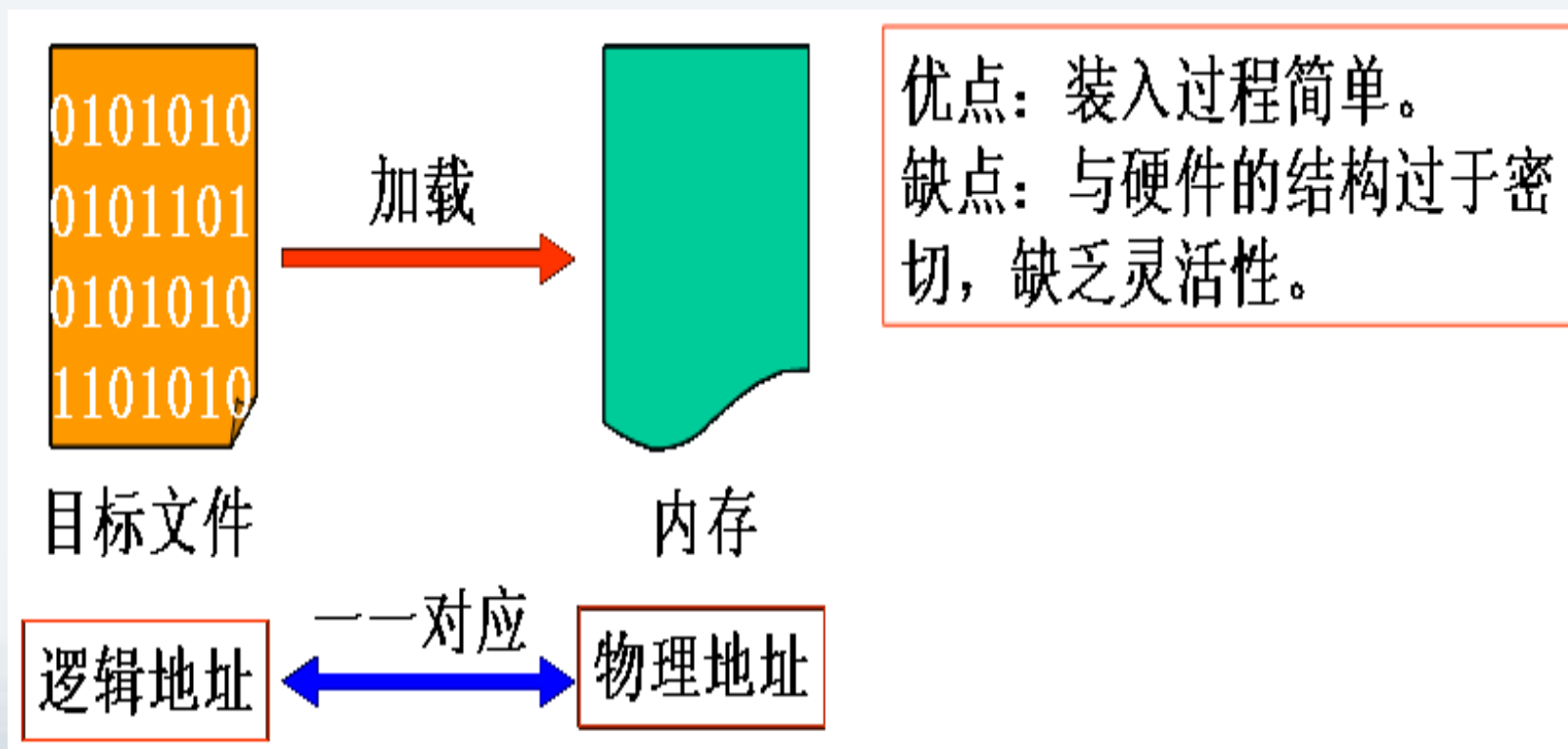
常见的地址重定位技术

1. 绝对装入(Absolute loading) / 固定地址再定位

- 程序的地址再定位是在程序执行之前被确定的，也就是在编译连接时直接生成实际存储器地址(物理地址)。在此，程序地址空间和内存地址空间是一一对应的。

绝对装入

- 例如：单片机，MS-DOS中.com格式程序。



常见的地址重定位技术

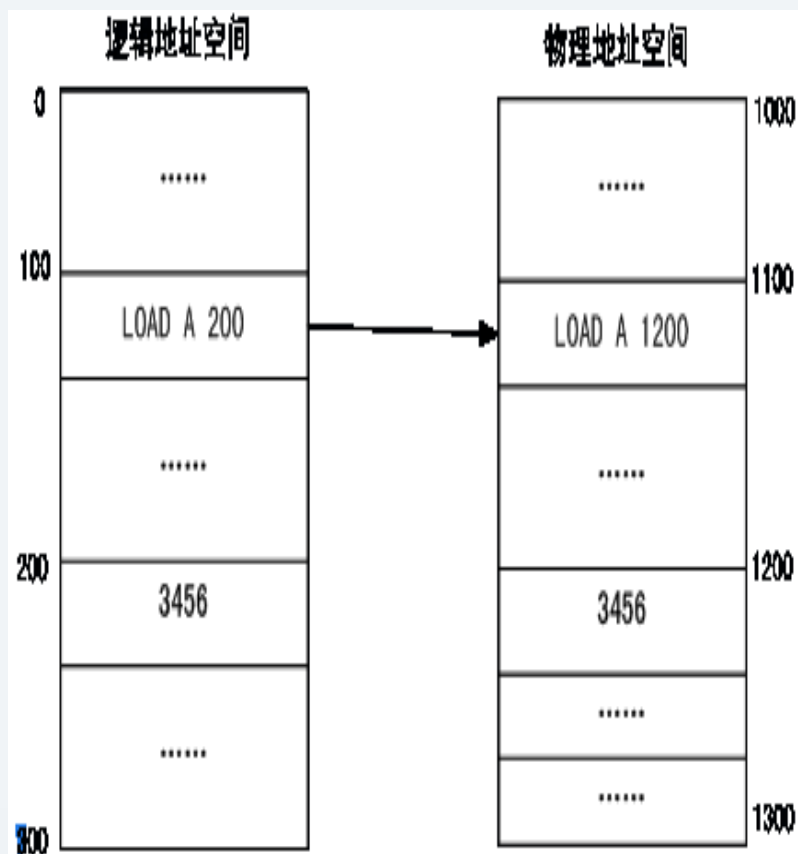
2. 可重定位装入(Relocatable Loading)

- 即指程序装入内存时，由于程序的逻辑地址和物理地址不一致，由逻辑地址到物理地址的映射过程。
 - 静态重定位
 - 动态重定位

静态重定位

- 在装入一个程序时，把程序中的指令地址和数据地址全部转换成绝对地址。由于地址转换工作是在程序开始**执行前**集中完成的，所以在程序执行过程中就无需再进行地址转换工作。

静态重定位



优点：无需硬件支持，容易实现。
早期的操作系统中大多数都采用这种方法。

缺点：必须分配连续的存储区域；
执行期间不能扩充存储空间，
也不能在内存中移动，内存利用率低，不便于共享。

动态重定位

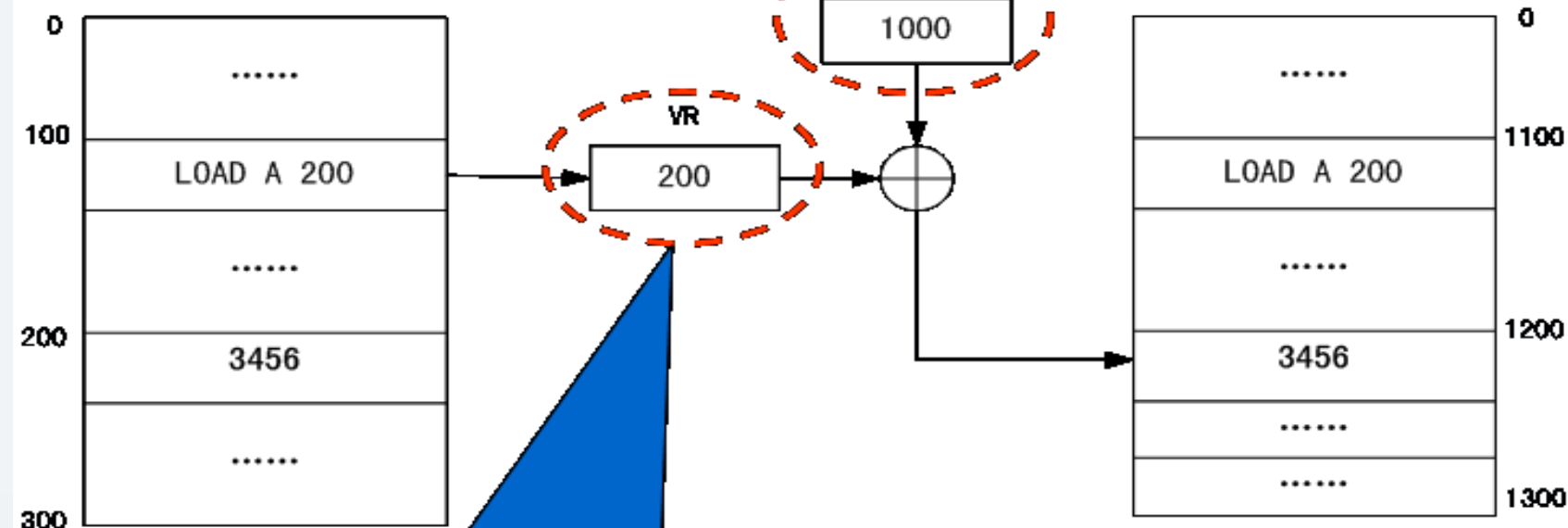
- 在装入程序时，不修改程序的逻辑地址值，而是直接把程序装入到分配的主存区域中。在访问物理内存之前，再实时地将逻辑地址转换成物理地址。

动态重定位

BR: 基址寄存器, 存放程序的起始地址

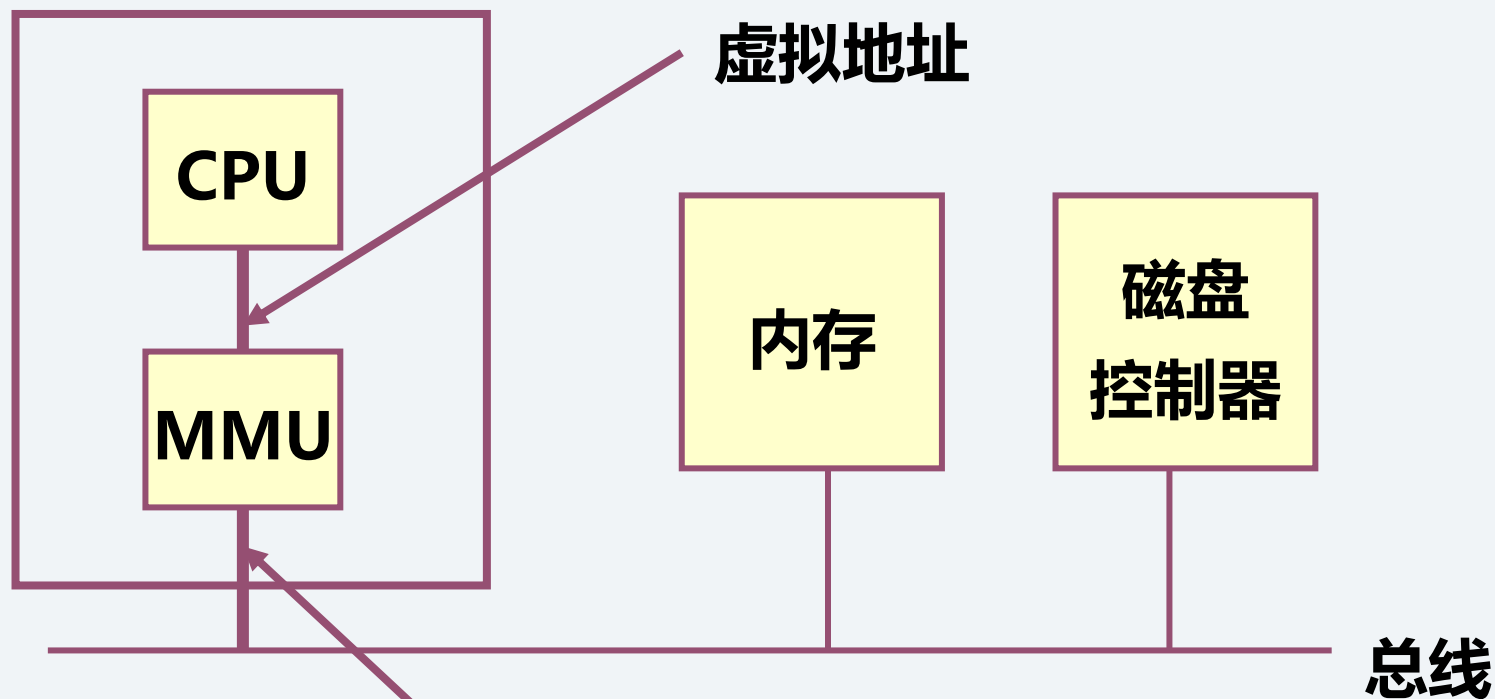
逻辑地址空间

物理地址空间



VR: 变址寄存器, 存放需要变换的逻辑地址

内存地址空间转换的硬件机制



MMU：内存管理单元

物理地址

动态重定位

优点：

- 程序在执行期间可以换入和换出内存，可以解决内存紧张状态；
- 可以在内存中移动——把内存中的碎片集中起来，可以充分利用空间；
- 不必给程序分配连续的内存空间，可以较好的利用较小的内存块；
- 若干用户可以共享同一程序，实现共享。

缺点：

- 需要附加的硬件支持，实现存储管理的软件算法比较复杂。

问题的提出

若给定计算机的内存大小，如何管理？预期目标：

- 高空间利用率？
- 尽可能快的访问速度？
- 超乎现实的容量？

3.1 基本存储管理



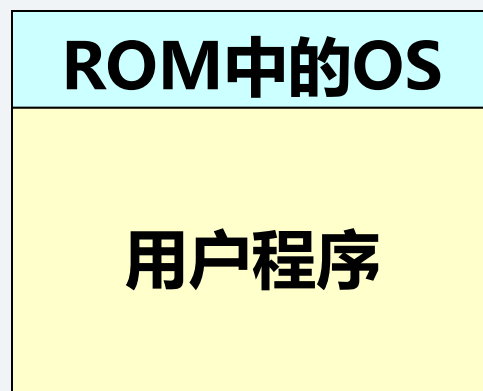
3.1.1 单道静态存储管理

- 最简单的管理方式（只有分配与回收）
- 操作系统和用户程序共享RAM
- 除了低端的嵌入式系统外，其他的计算机不再使用这种方式

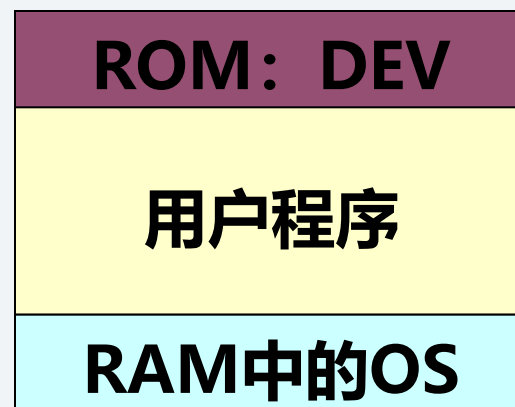
单道静态存储管理



早期大型机
使用的内存
管理方式



少数掌上电脑
和嵌入式系统
使用的内存管
理方式



早期PC使用
的内存管理方
式 (MS-
DOS)

单道静态存储管理

- 通常同一个时刻只能有一个进程在运行。一旦用户键入了一个命令，操作系统就把需要的程序从磁盘复制到内存中并执行；当进程运行结束后，操作系统在用户终端显示提示符并等待新的命令。收到新的命令后，它把新的程序装入内存，覆盖前一个程序。

问题的提出

如何支持多道程序？

分区存储管理方案

- 思想：把内存分为一些大小相等或不等的分区(Partition)，装入时每个应用程序占用一个或几个分区，操作系统占用其中一个分区。适用于多道程序系统和分时系统，支持多个程序并发执行。
 - 固定分区管理
 - 可变分区管理

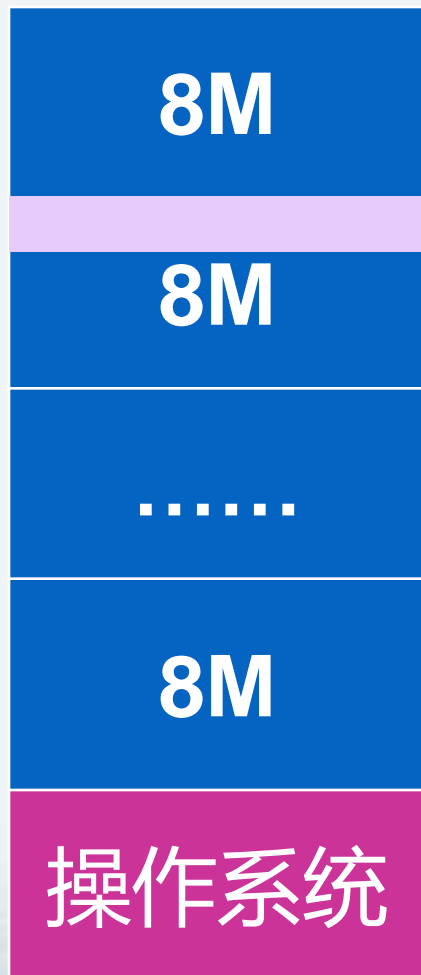
3.1.2 固定分区多道存储管理

将内存的用户空间划分为若干个固定大小的连续分区，
在每个分区中只装入一道作业。

1. 划分分区的方法：

- 分区大小相等：只适合于多个相同程序的并发执行（处理多个类型相同的对象）。
- 分区大小不等：多个小分区、适量的中等分区、少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。

分区大小相等



} 内部碎片 internal fragmentation

- 大程序放不下
- 小程序浪费空间
- 分区数目固定

分区大小不等

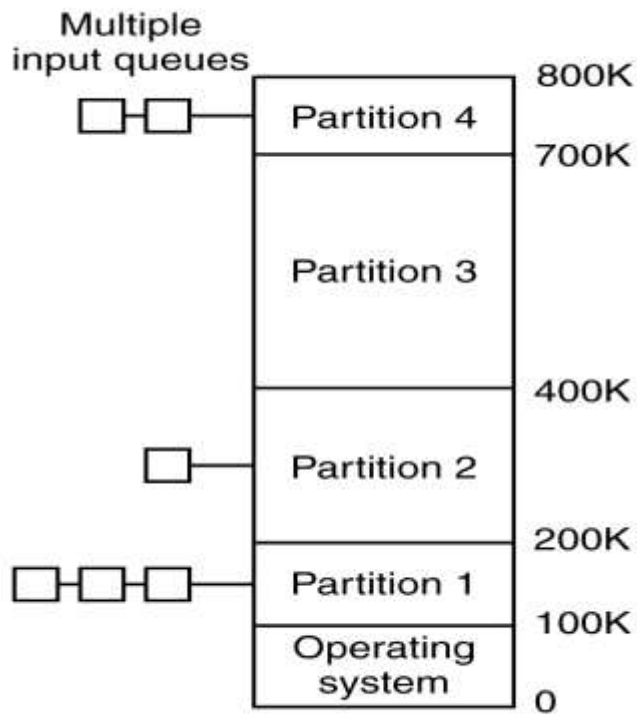


- 分区数目固定
- 内部碎片

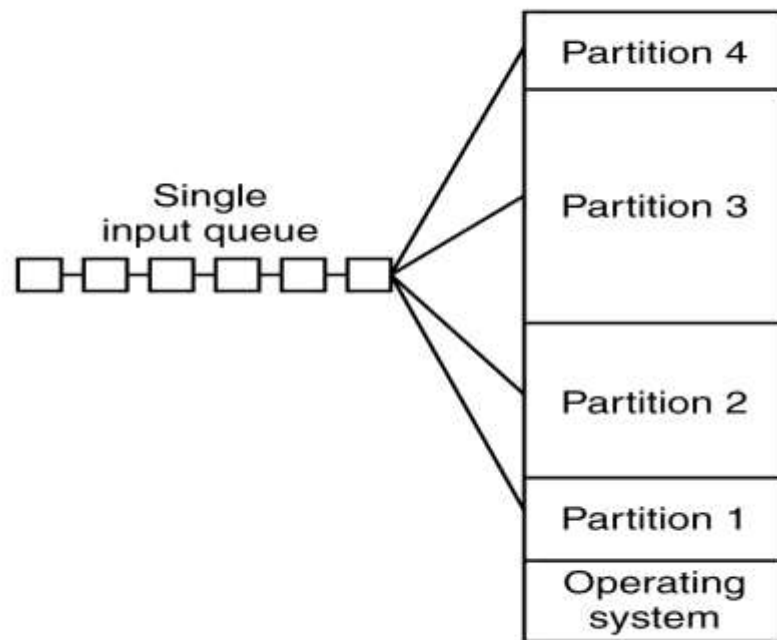
问题的提出

- 固定分区多道存储管理方式下，如何完成内存分配？

固定分区中的内存分配策略



(a)



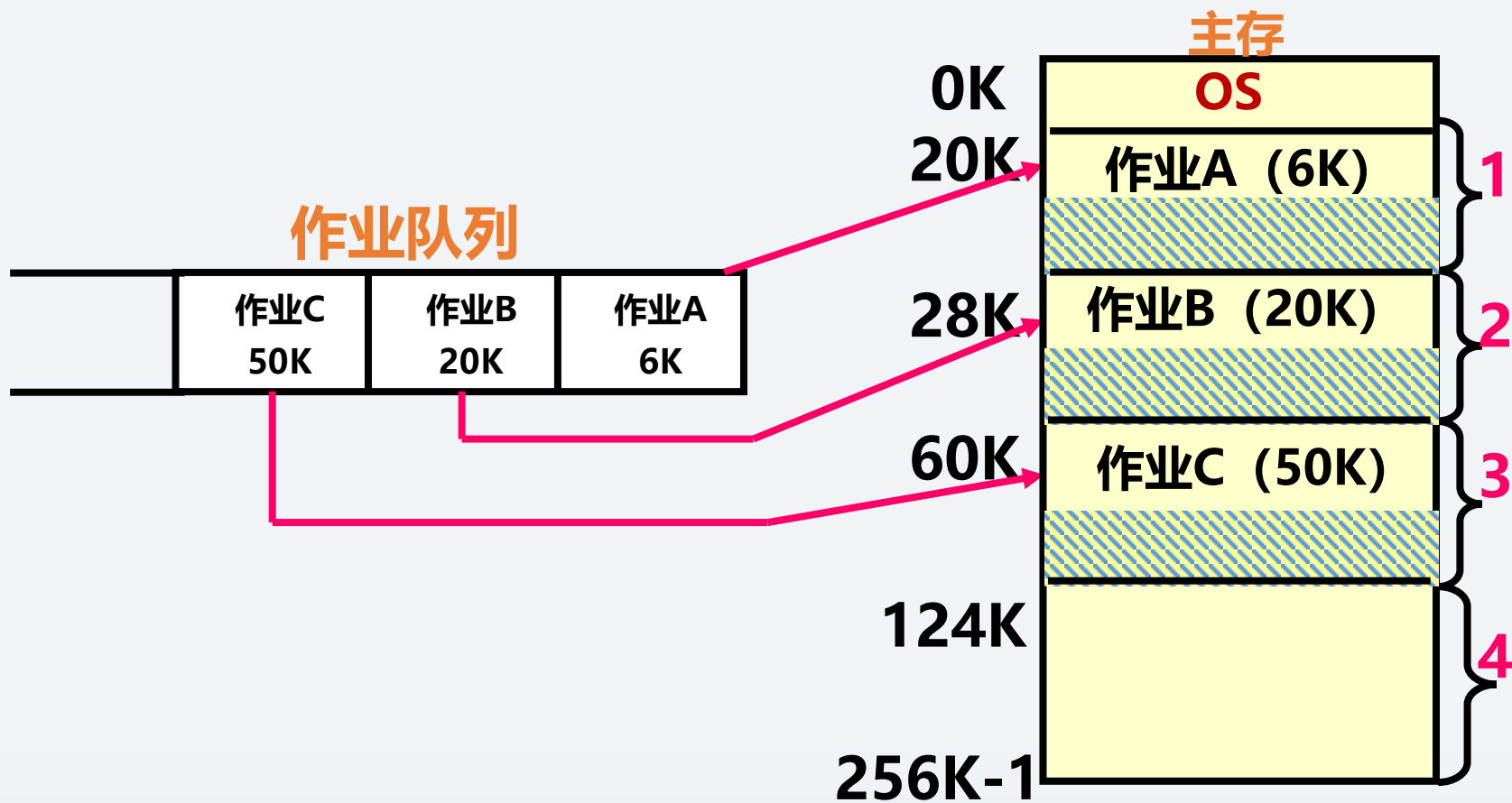
(b)

某个分区可能长期空闲

2. 内存分配

- 分区使用表：用于记录分区的大小和使用情况，按分区大小排队。包括每个分区的起始地址、大小和状态（是否分配）。
- 用户程序需要装入时，内存分配程序检索该表，找出一个能满足要求尚未分配的分区，分配给该程序，并将其表项中的状态置为“已分配”。
- 若未找到大小足够的分区，则拒绝为用户程序分配内存。

例：某系统的内存容量为256K，操作系统占用低地址的20K，其余空间划分成4个固定大小的分区。



分区说明表

分区号	大小(KB)	始址(KB)	状态
1	8	20	已分配
2	32	28	已分配
3	64	60	已分配
4	132	124	未分配

3.1.2 固定分区多道存储管理

特点：

- 内存中同时可以容纳多道程序；
- 程序必须连续存放，且要一次全部装入。

3.1.2 固定分区多道存储管理

优点：

- 比单一连续分配方法，内存的利用率提高了；
- 可以支持多道程序；
- 实现简单，开销小。

3.1.2 固定分区多道存储管理

缺点：

- 作业必须预先能够估计自己要占用多大的内存空间，有时候这是难以做到的；
- 存在内碎片，造成存储空间的浪费；
- 分区总数固定，限制了并发执行的程序数目。

静态存储管理小结

- 静态存储管理的特点
 - 内存容纳的进程数是静态不变的
 - 每个进程的内存大小是静态不变的

动态存储管理导论

- 动态存储管理的出发点
 - 物理内存可容纳的进程数动态可变
 - 每个进程占用的内存大小动态可变
 - 超出物理内存大小的进程可以运行

动态存储管理导论

两种方法

- 交换：整体上调入进程，使该进程运行一段时间，再存回磁盘。
- 虚拟存储器：能使程序在只有一部分被调入主存的情况下运行。

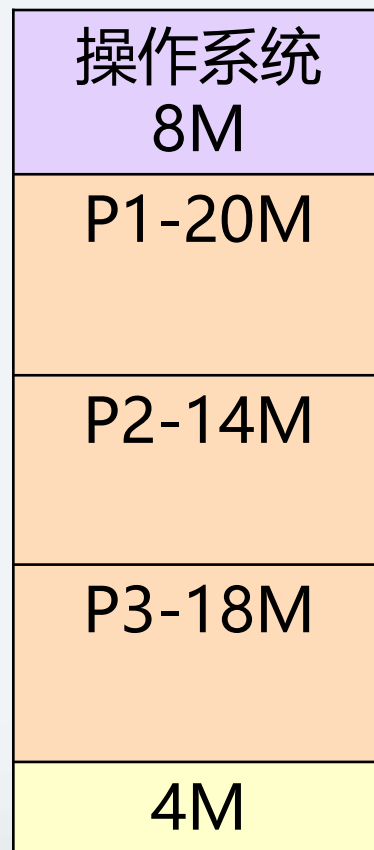
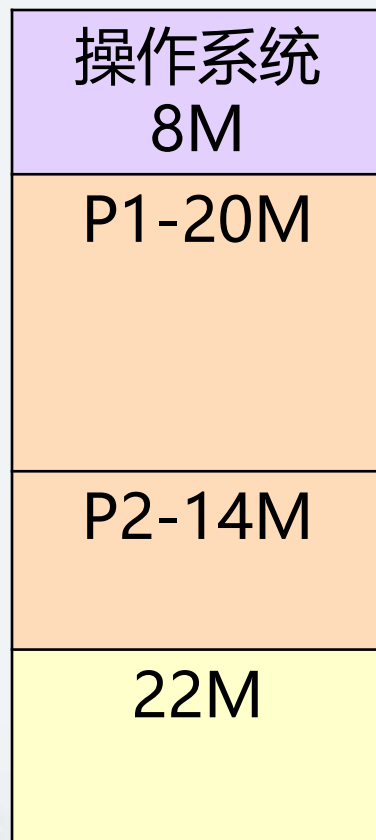
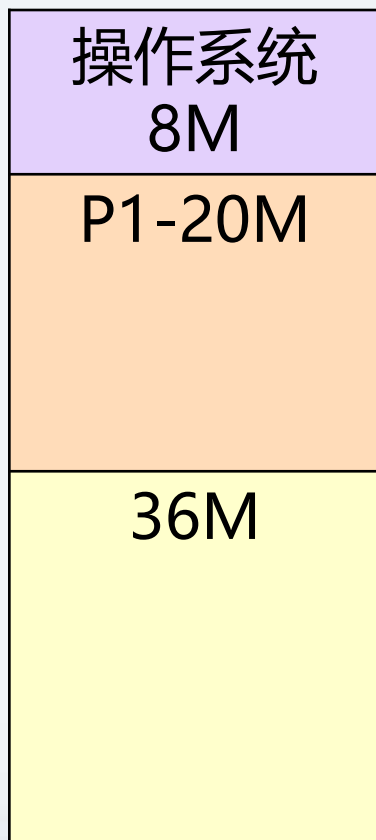
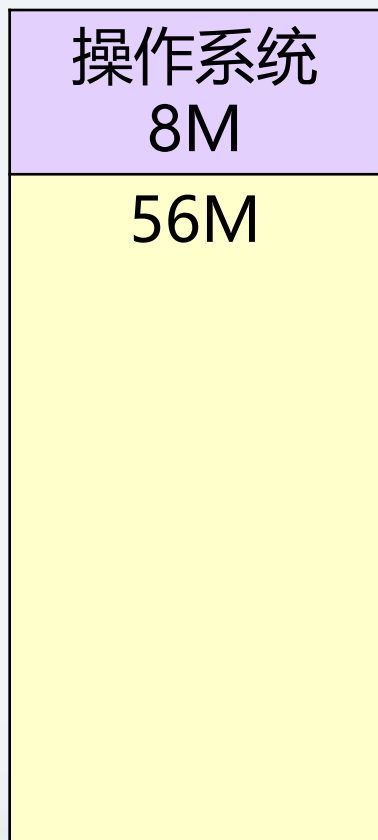
3.2 交换 (Swapping)

基于交换的动态存储管理

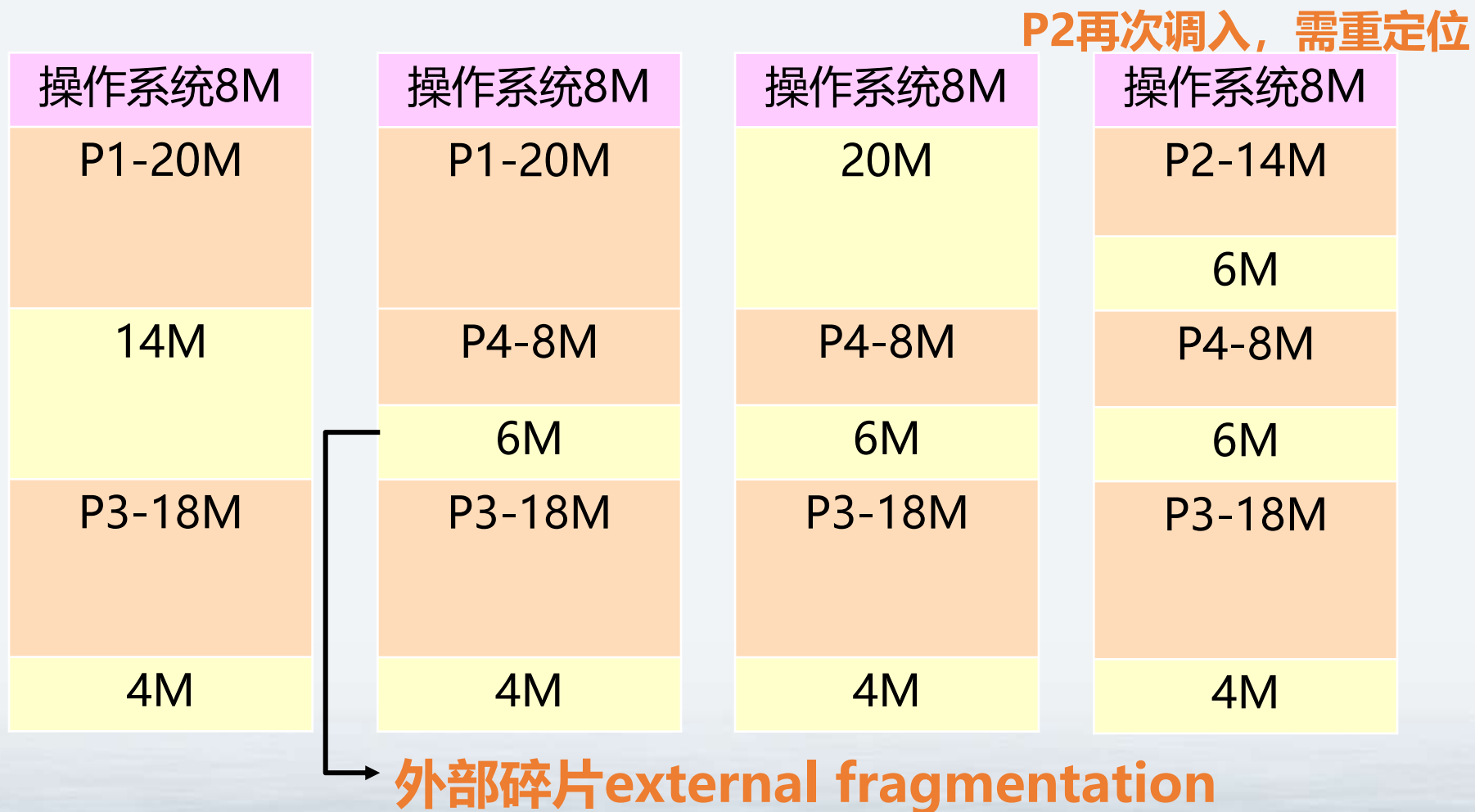
- 所有可运行的进程必须小于内存空间
- 当进程处于阻塞或就绪时交换到磁盘
- 进程在运行过程中将多次进出内存（需重定位）

动态分区

0



动态分区



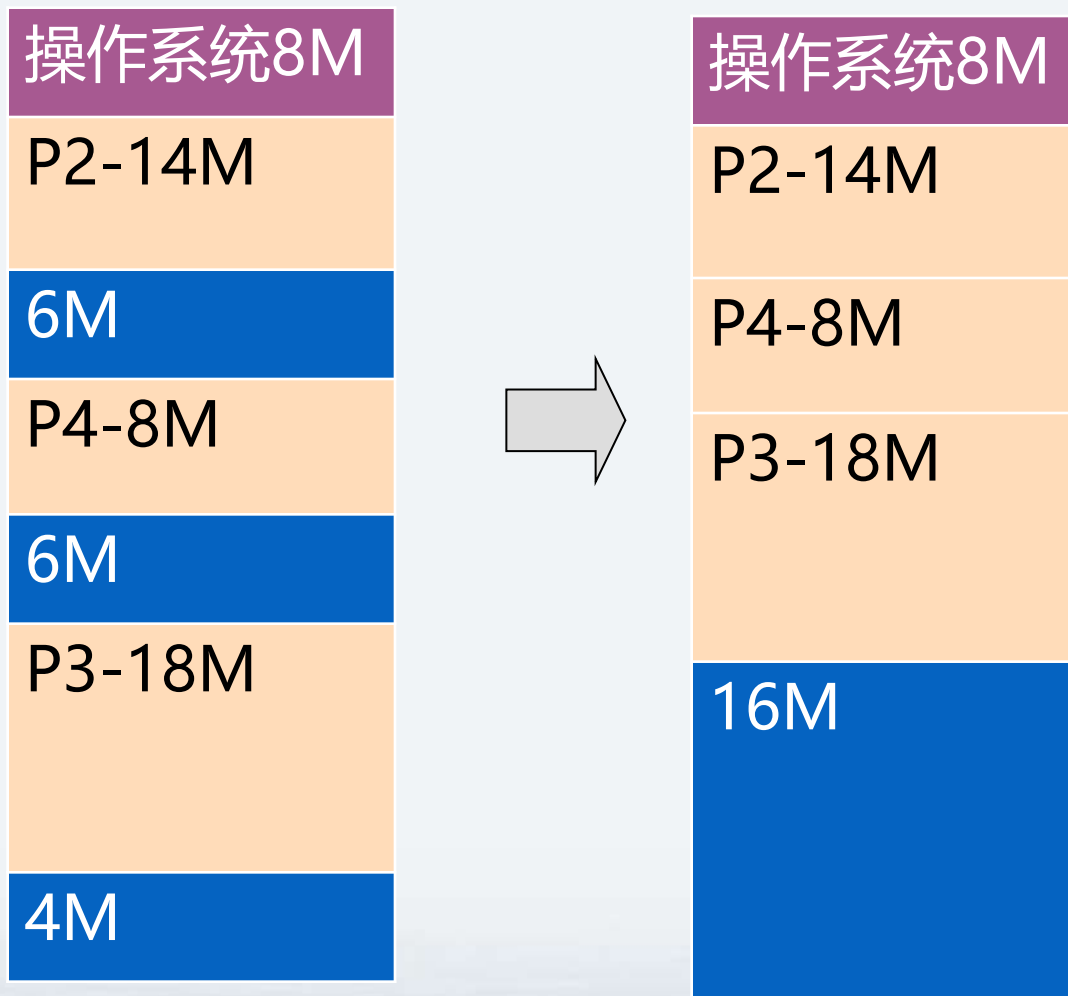
动态分区

- 分区的数量、位置、大小随着进程的出入是动态变化的。
- 不再受到可能太大或太小的固定分区的约束，内存利用率提高。

动态分区

- 不能被利用的小分区称为“零头”或“碎片”。
- 通过移动，把多个分散的小分区拼接成大分区的方法被称为紧缩(compact) 。
- 每次“紧缩”后必须对移动了的程序或数据进行重新定位。

紧缩(compaction)

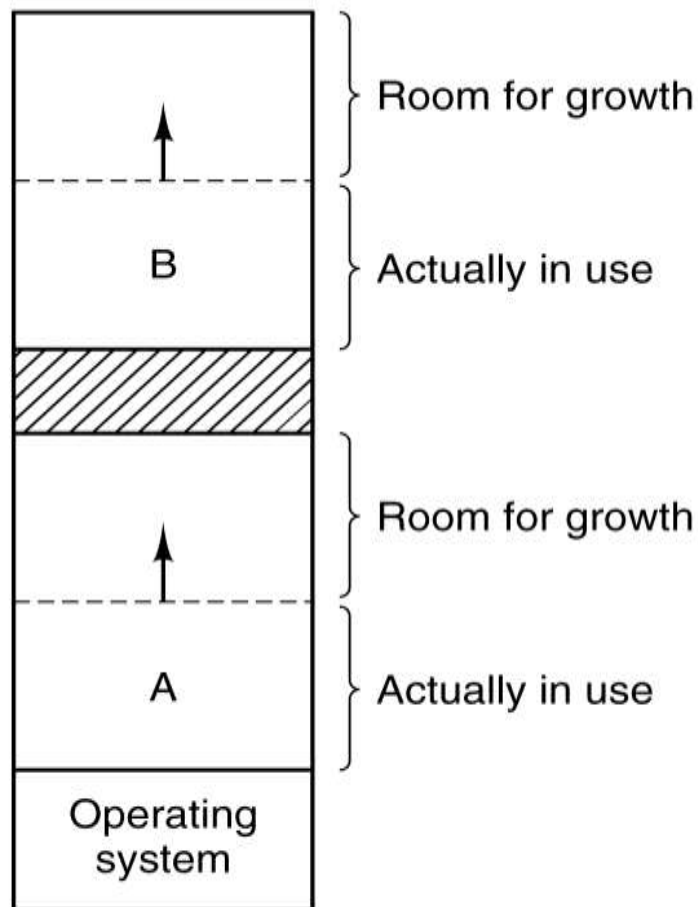


- 浪费时间
- 动态重定位

如何满足进程的动态增长

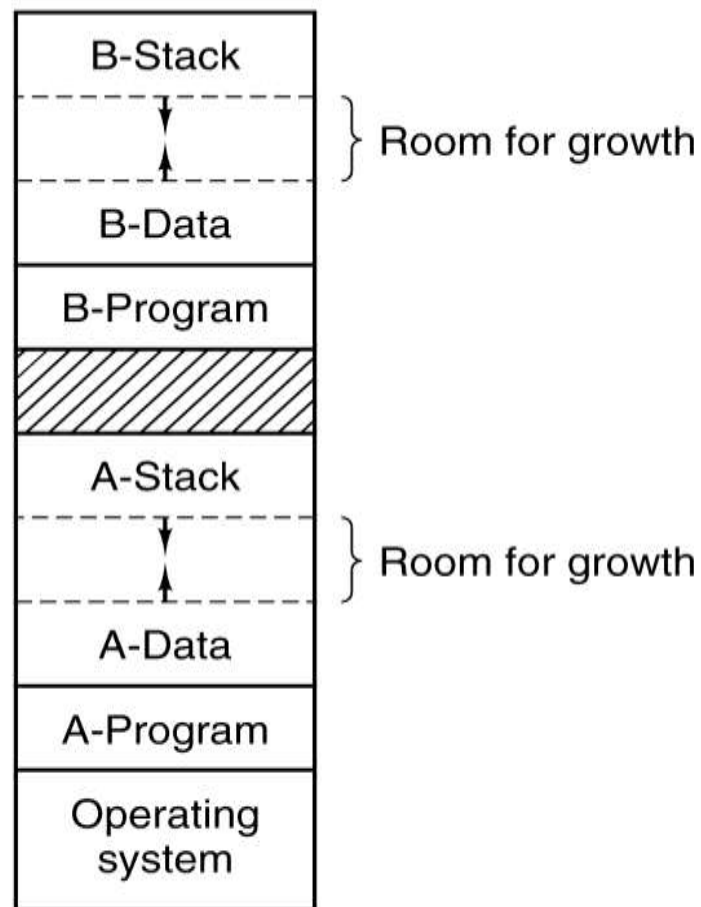
- 采用交换思想满足内存增长
- 采用预留内存空间的做法

为可能增长的数据段预留空间



(a)

为可能增长的数据段和堆栈段预留空间



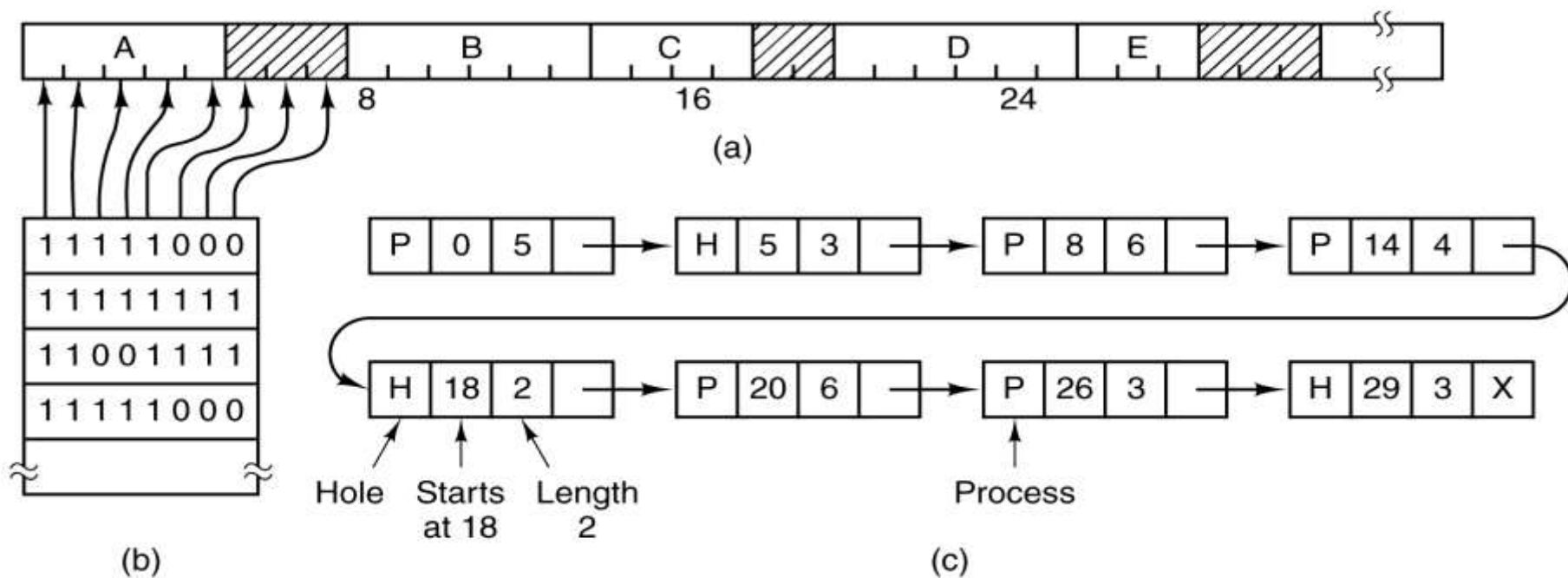
(b)

空闲内存管理

1. 使用位图的存储管理

- 用一串二进制位反映内存空间的分配使用情况, 每个分配单位对应一位, “1”表示占用(已分配), “0”表示空闲(未分配)。
- 申请内存时, 可以在位图中查找为0的位
- 归还时, 将对应位置0

使用位图的存储管理



位图存储管理的设计与实现

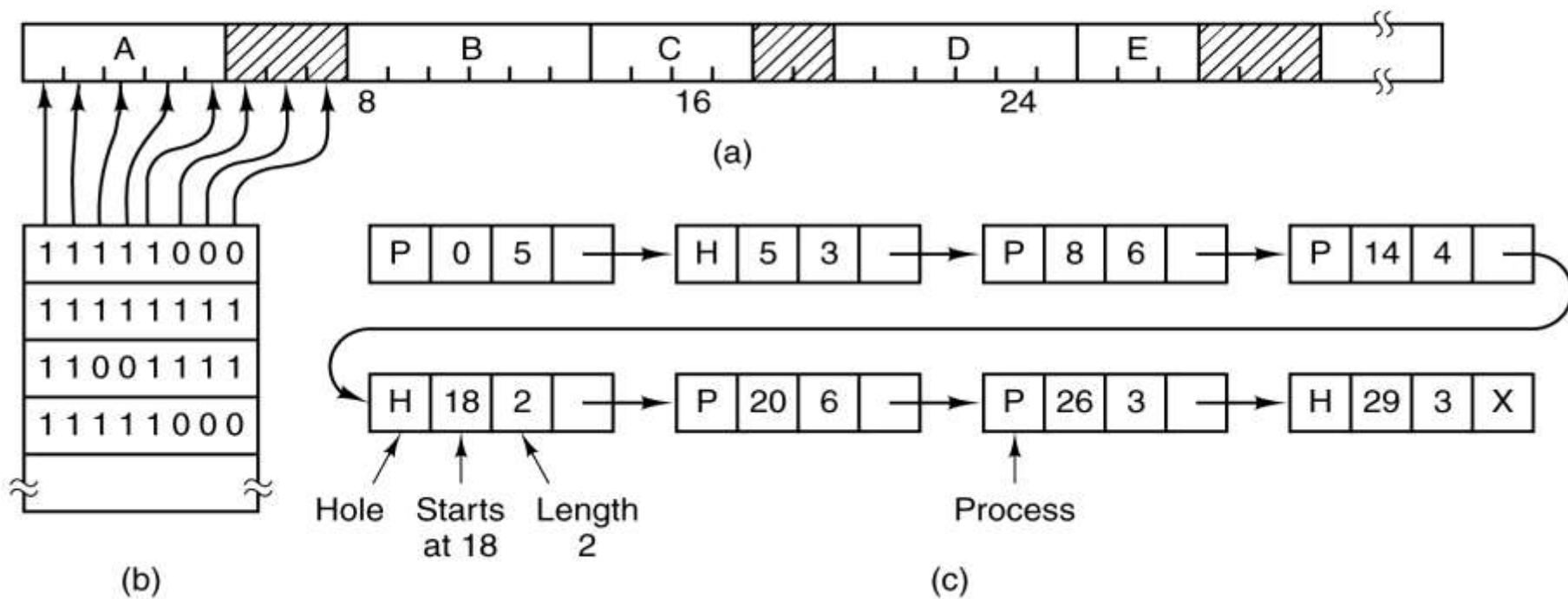
- 关键问题1： 位图的大小。首先要确定内存分配的基本单位的大小。
- 关键问题2： 如何实现内存分配？ 分配的过程类似于在字符数组中找出连续的0/1串。

空闲内存管理

使用链表的存储管理

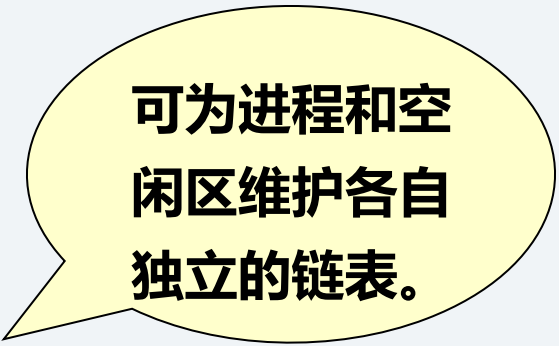
- 不确定内存分配基本单位，而是描述内存占用情况
- 设定内存区段的数据结点，按照地址排序形成链表
- 基于链表的搜索来实现内存分配

3.2.2 使用链表的存储管理



使用链表的存储管理

- 链表存储管理的设计与实现
 - 关键问题1：设定几个链表？
 - 关键问题2：如何确定分配策略？



可为进程和空闲区维护各自独立的链表。

使用链表的存储管理

- 首次适应算法：将地址最小的够用的空间分配出去
- 下次适应算法：从上次分配位置开始搜索，将地址最小的够用的空间分配出去
- 最佳适应算法：将够用的长度最小的空间分配出去
- 最差适应算法：将够用的长度最大的空间分配出去

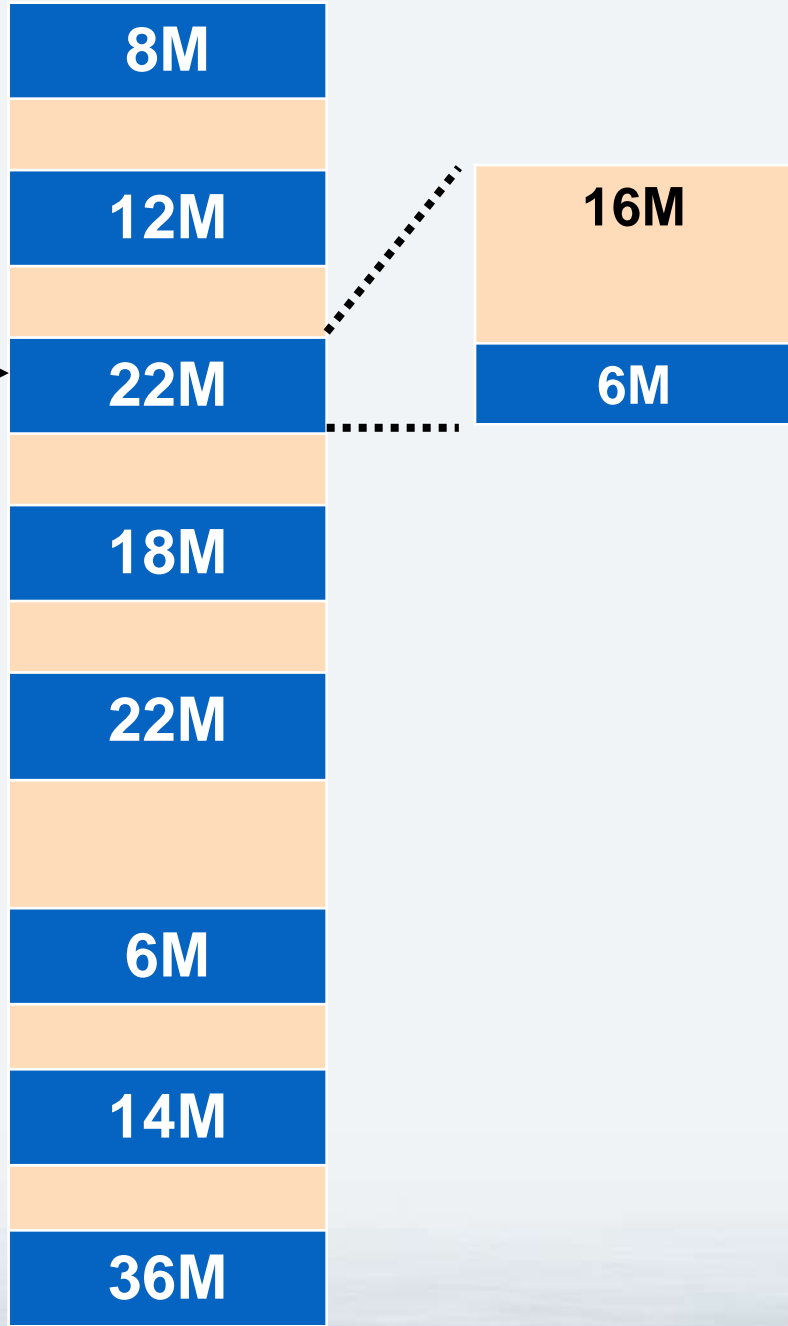
首次适应法

- 要求空闲区按首址递增的次序组织空闲区表（队列）。
- 当进程申请大小为SIZE的内存时，系统从空闲区表的第一个表目开始查询，直到首次找到等于或大于SIZE的空闲区。从该区中划出大小为SIZE的分区分配给进程，余下的部分仍作为一个空闲区留在空闲区表中，但要修改其首址和大小。

0地址

首次适应 →

申请空间大小为16M



首次适应法

优点：

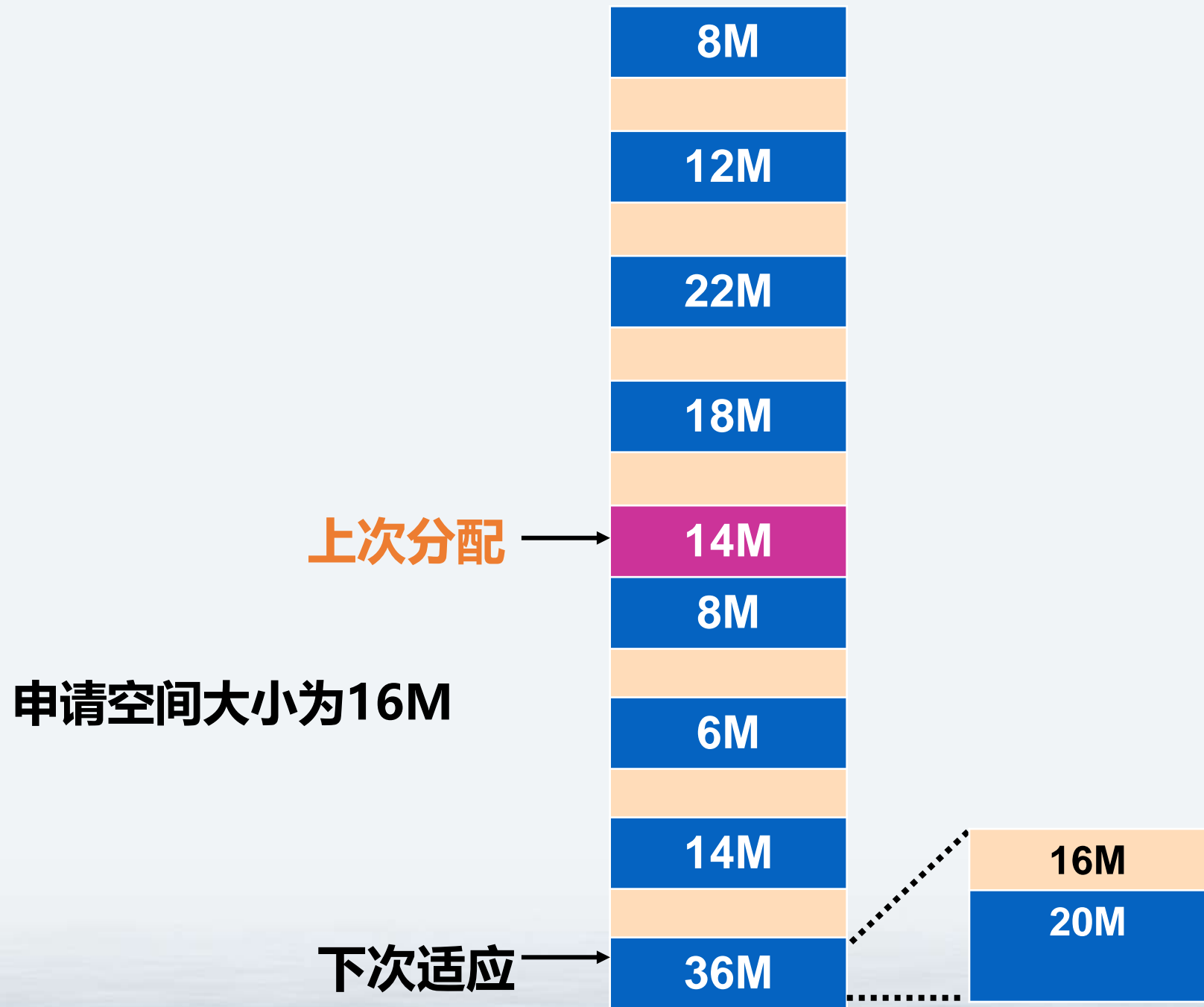
- 该算法是尽可能地利用低地址空间，从而保证高地址空间有较大的空闲区。

缺点：

- 低地址部分的不断划分，会留下许多难以利用的、很小的空闲分区，而每次查找都是从低地址部分开始，会增加查找可利用分区时的开销。

下次适应算法

- 将空闲区组织成一个循环链，每次为存储请求查找合适的分区时，总是从上次查找结束的地方开始，只要找到一个足够大的空闲区，就将它划分后分配出去。
- 优点：能使内存中的空闲分区分布均匀，从而减少了查找空闲分区时的开销。
- 缺点：使内存中缺乏大的分区。



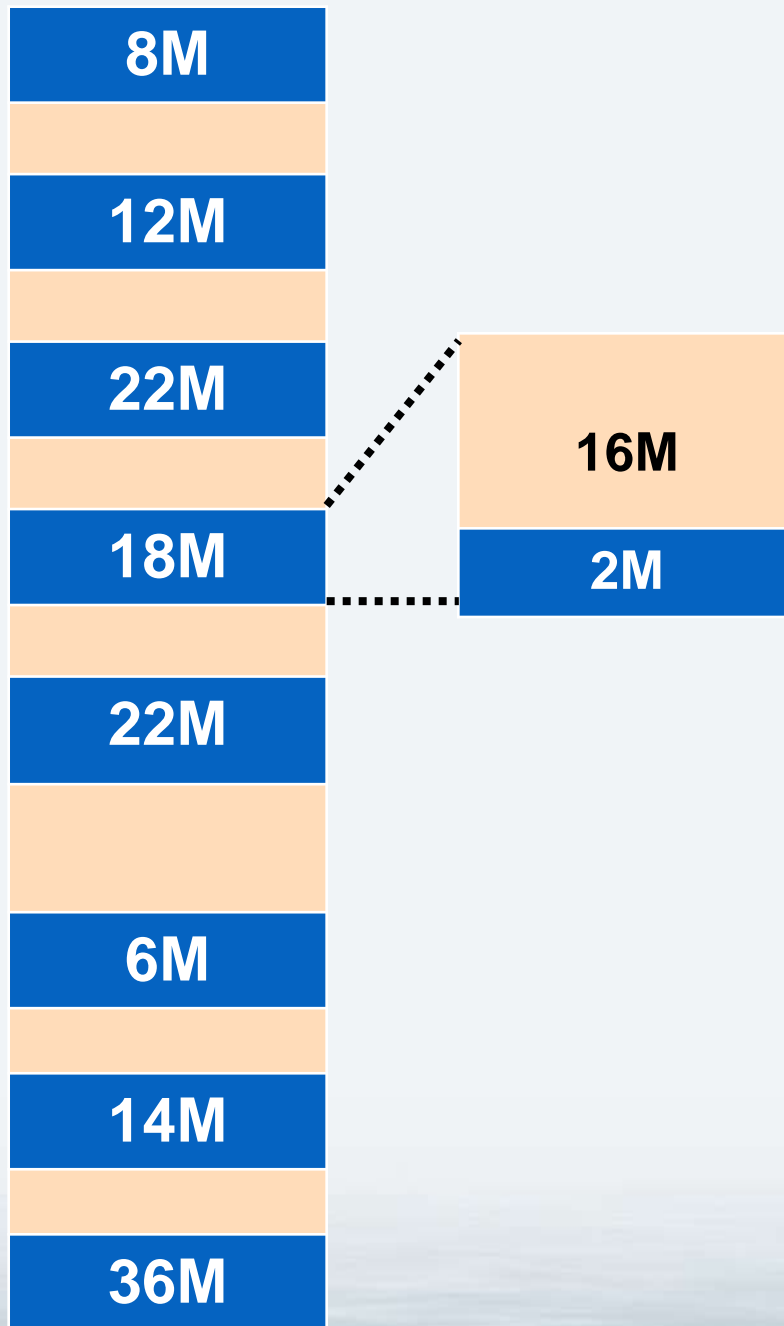
最佳适应算法

- 将空闲区链表中一个大小不小于“请求”且最接近“请求”的空闲块的一部分分配给用户。
- 分配时需要对空闲区链表从头至尾查询一遍。为了避免每次分配都要查询整个链表，通常要求节点从小到大排序，由此只需找到第一个足够大的空闲块即可予以分配。

0地址

最佳适应 →

申请空间大小为16M



最佳适应算法

优点：

- 选中的空闲区是满足要求的最小空闲区，而不致于毁掉较大的空闲区。

缺点：

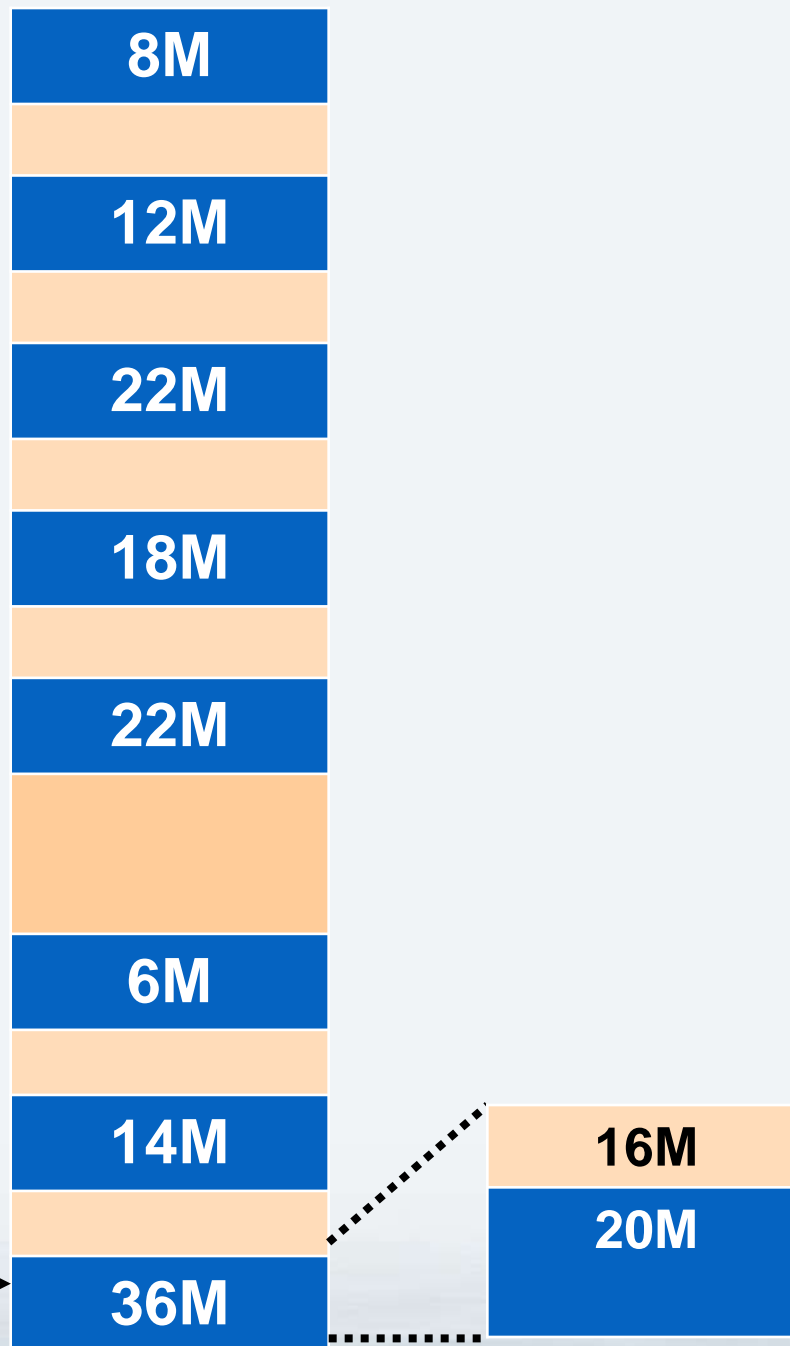
- 空闲区的大小一般与申请分区大小不相等，因此将其一分为二，留下来的空闲区一般情况下是很小的，以致无法使用。随着时间的推移，系统中的小空闲区会越来越多，从而造成存储区的大量浪费。

最差适应算法

- 按大小递减的顺序形成空闲区链，分配时直接从空闲区链的第一个空闲分区中分配（不能满足需要则不分配）。
- 这种算法将使链表的节点大小逐渐趋于均匀。

申请空间大小为16M

最差适应 →



最差适应算法

最坏适应法看起来似乎有些荒唐，但在更加严密地考察后，还是有它的优点：

- 当程序装入内存中最大的空闲区后，剩下的空闲区还可能相当大，还能装下较大的程序。
- 每次仅作一次查询工作。

几点说明：

- 上述几种分配算法各有利弊，到底哪种最好不能一概而论，而应针对具体作业序列来分析。
- 对于某一作业序列来说，某种算法能将该作业序列中所有作业安置完毕，那么我们说该算法对这一作业序列是合适的。
- 对于某一算法而言，如它不能立即满足某一要求，而其它算法却可以满足此要求，则这一算法对该作业序列是不合适的。

【例】

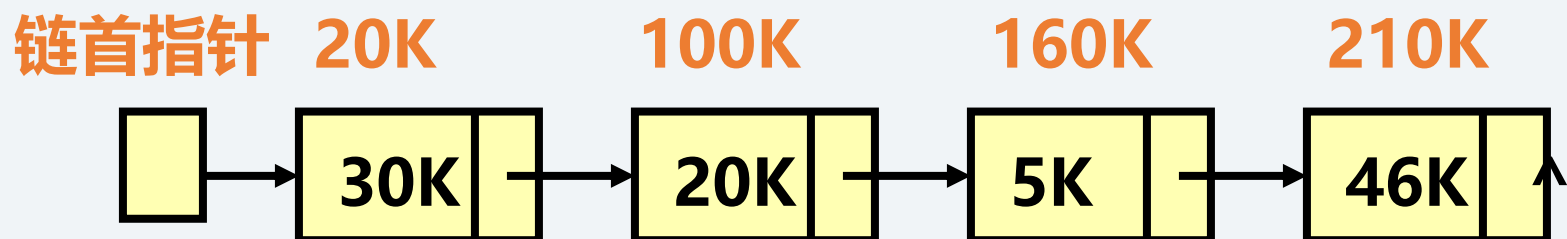
有一作业序列：作业A要求18K；作业B要求25K，作业C要求30K。

分析系统中空闲区，并按首次适应，下次适应，最佳适应三种算法组成空闲区队列。

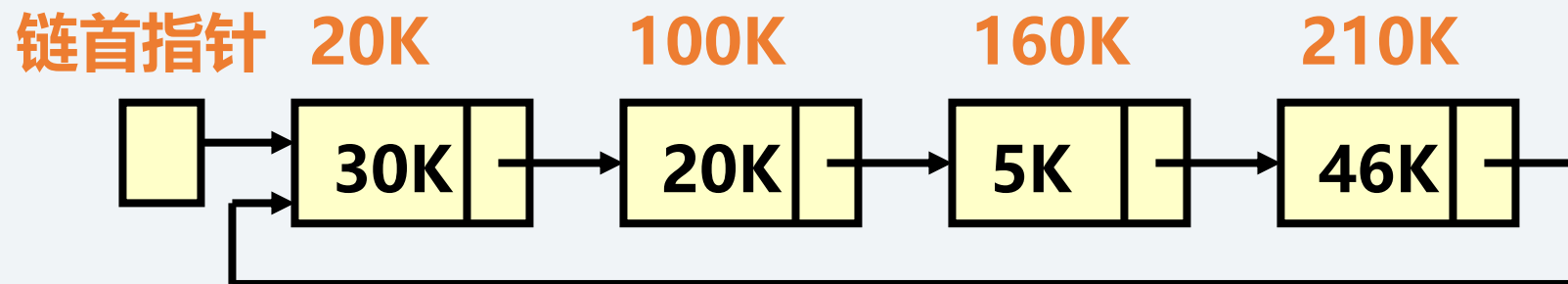
图中灰色部分为空闲区。



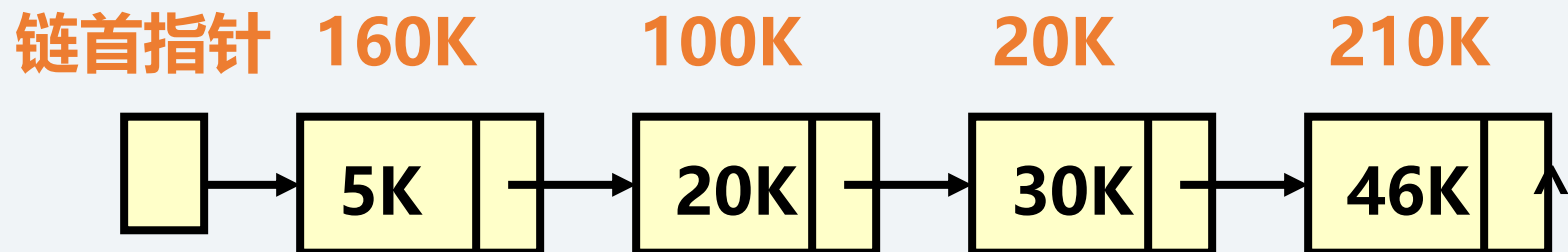
① 首次适应算法的空闲区队列：



② 下次适应算法的空闲区队列：



③ 最佳适应算法的空闲区队列：

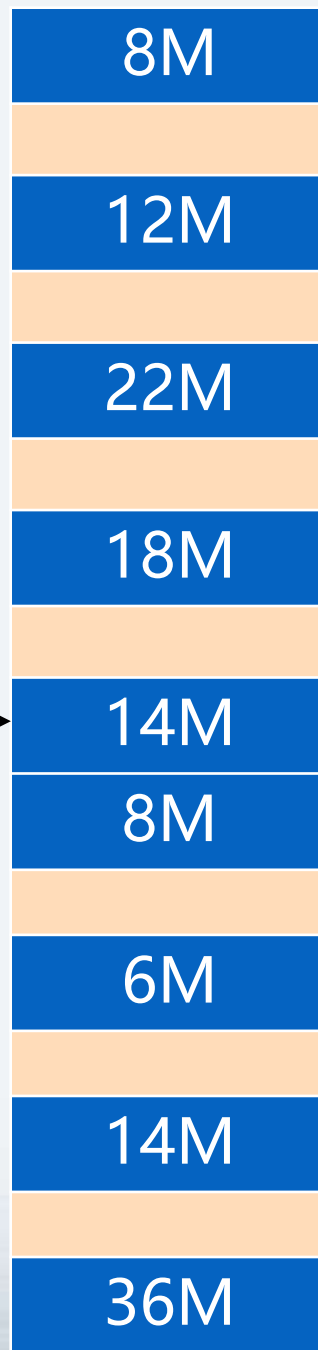
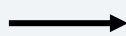


分配算法对比

首次适应算法	保证高地址空间有较大的空闲区；但地址低端会有大量碎片
下次适应算法	空闲区均匀分布 没有大的空闲块
最佳适应算法	很多碎片 经常需要紧缩
最差适应算法	碎片较少 没有大的空闲块

分区的回收

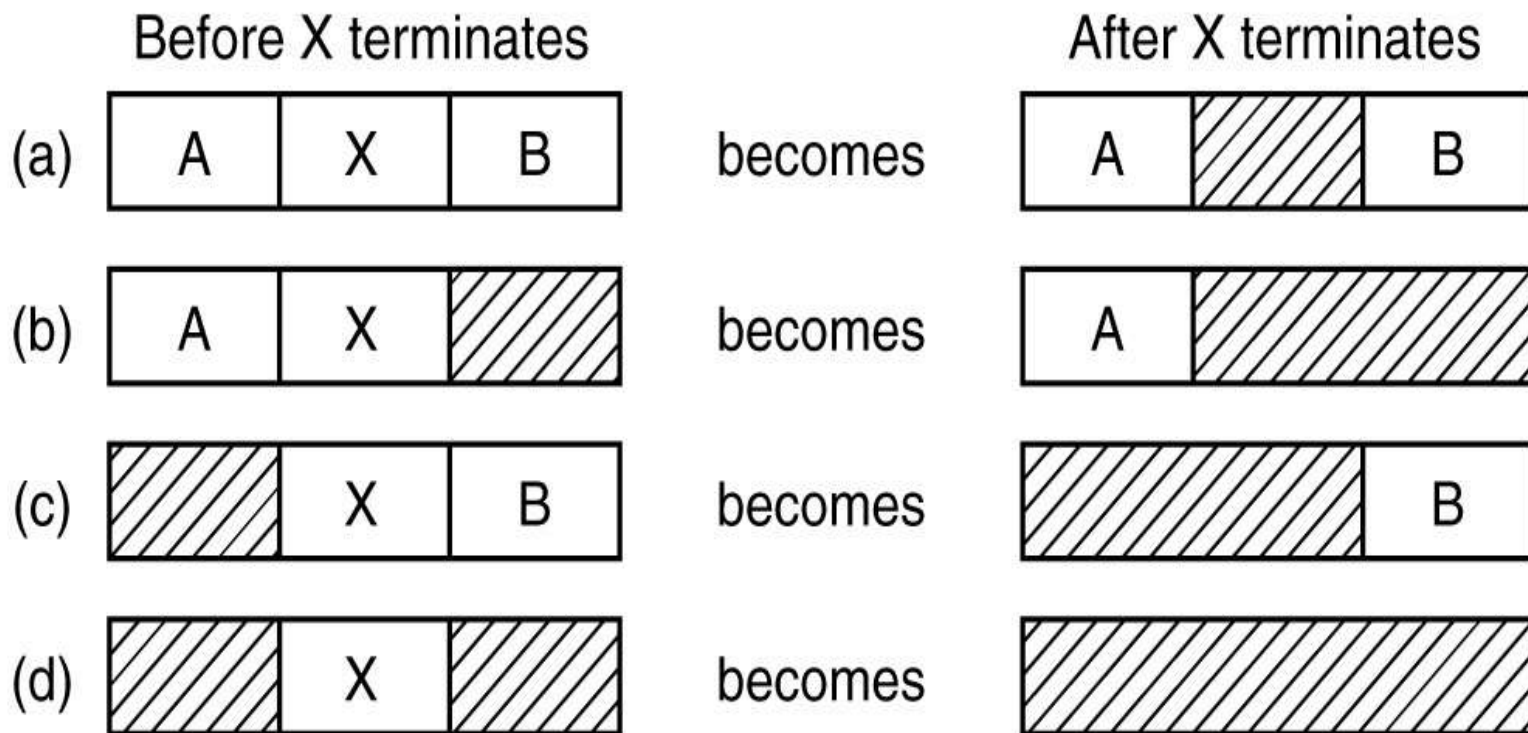
上次释放



分区的回收

- 当某个进程释放某存储区时，系统首先检查释放区是否与系统中的空闲区相邻，若相邻则把释放区合并到相邻的空闲区中去，否则把释放区作为一个空闲区插入到空闲区表的适当位置。

分区的回收



静态存储管理与交换技术对比

	单道独占	固定分区	交换技术
进程大小	小于内存总量	小于分区容量	小于可用内存量
内存分配方式	静态	静态	动态
多道程序	不支持	支持	支持
内存空间分布	连续	连续	连续
内存动态变化	不支持	不支持	支持

3.3 虚拟存储器

- 静态存储和交换技术的缺点：无法有效支持比物理内存空间大的程序的运行。
- 常规存储器管理方式的特征：
 - 一次性:作业在运行前需一次性全部载入内存
 - 驻留性：作业装入内存后，便一直驻留在内存中，直至作业运行结束

3.3 虚拟存储器

if(x > 5) A();

else B();

- A()与B()不会同时运行。

3.3 虚拟存储器

- 虚拟存储：用较小的空间运行较大的程序
- 为什么会产生虚拟存储？——时间换空间
- 用户在编程时可以不考虑实际内存的大小，认为自己编写多大程序就有多大的虚拟存储器对应。

虚拟存储的核心——局部性原理

- 程序运行的时间局部性
 - 一个内存单元被访问后，有可能在较短时间间隔内被重复访问
 - 时间局部性的体现：循环、重复调用
- 程序运行的空间局部性
 - 一个内存单元被访问后，其相邻单元在一定时间内也会被访问
 - 空间局部性的体现：数组、链表等

虚拟存储的核心——局部性原理

- 虚拟存储的核心思想
 - 基于局部性原理，将大程序划分为多个较小的部分
 - 当进程运行时，先将一部分程序装入内存，另一部分暂时留在外存，当要执行的指令不在内存时，由系统自动完成将它们从外存调入内存工作

3.3.1 分页

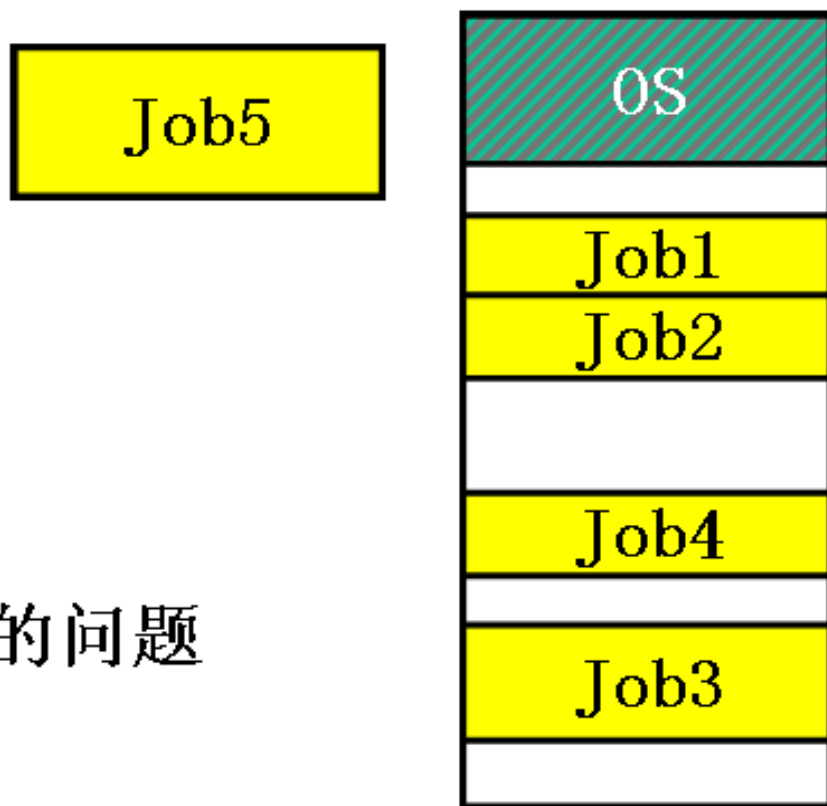
分区存储管理方案

要求作业存储时必须连续存放



页式存储管理方案

解决作业不连续存放的问题



3.3.1 分页

1. 页面

- 页（页面） - 把每个进程逻辑地址空间划分成若干大小相等的块，并为各页加以编号。
- 页帧 - 把内存空间划分成与页相同的块。

3.3.1 分页

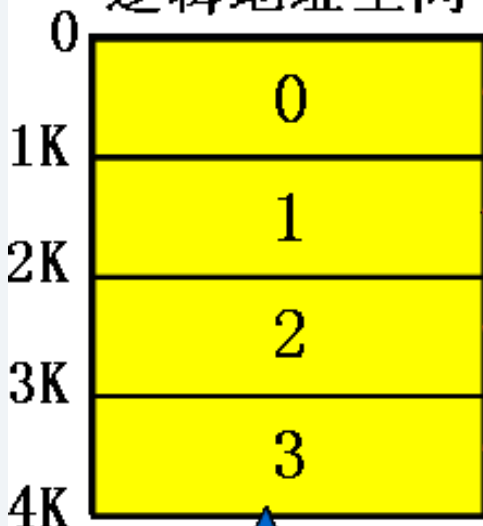
- 在为进程分配内存时，以页为单位将进程中的若干个页分别装入到多个可以不相邻的页帧中。

基本原理

实页/
主页

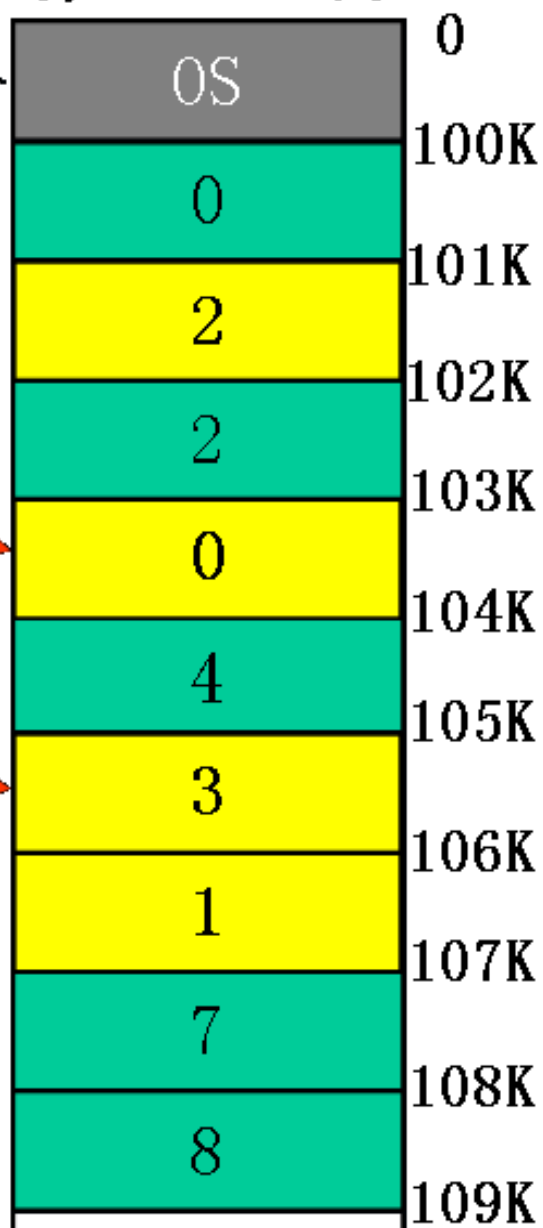
物理地址空间

逻辑地址空间



Q: 如何记录和管理这种映射关系?

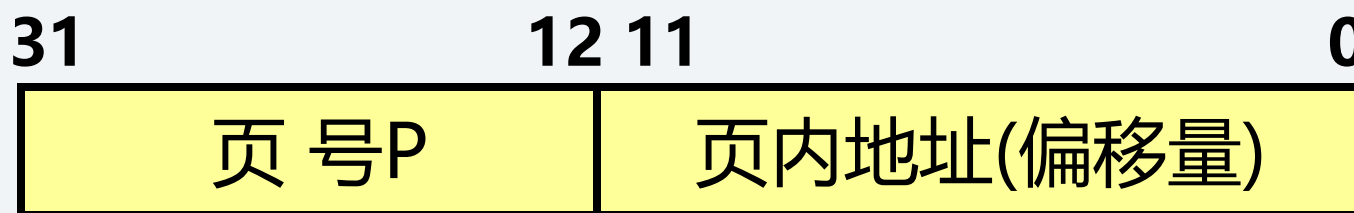
虚页: 大小相同, 常为2的整数幂。



3.3.1 分页

2. 分页式存储器的逻辑地址结构

- 分页地址中的地址结构确定了主存储器的分页大小,也决定了页面大小。



总页数: $2^{20} = 1\text{M}(\text{页})$; 每页大小为: $2^{12} = 4\text{KB}$

3.3.1 分页

- 对某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：

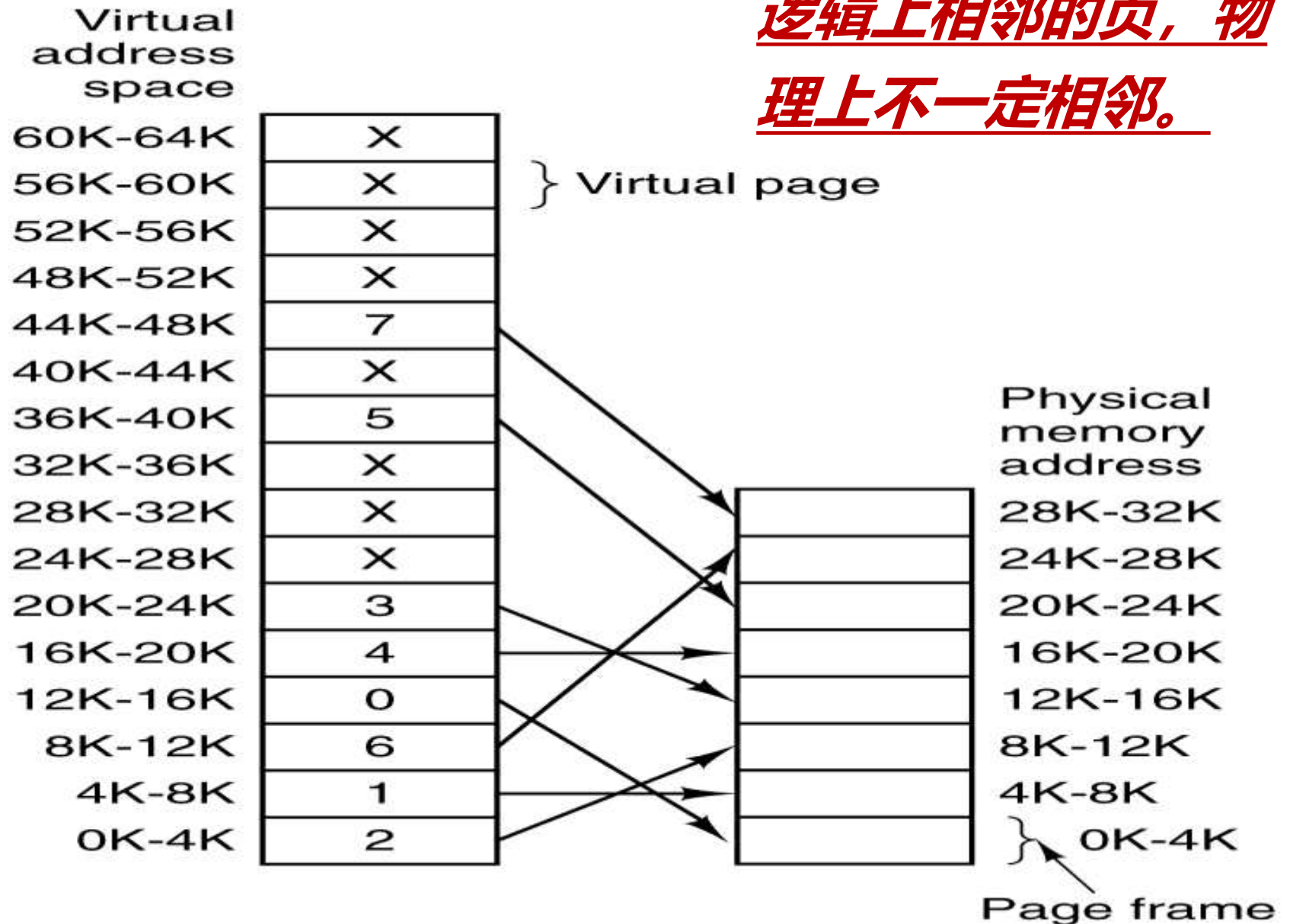
$$P = INT \left[\frac{A}{L} \right]$$

$$d = [A]MODL$$

3.3.1 分页

- 当逻辑地址空间大于物理地址空间时，必然有一些逻辑页没有建立与物理页帧的映射，当运行到这些逻辑页内的指令时，就必须将其调入物理内存。

逻辑上相邻的页，物理上不一定相邻。



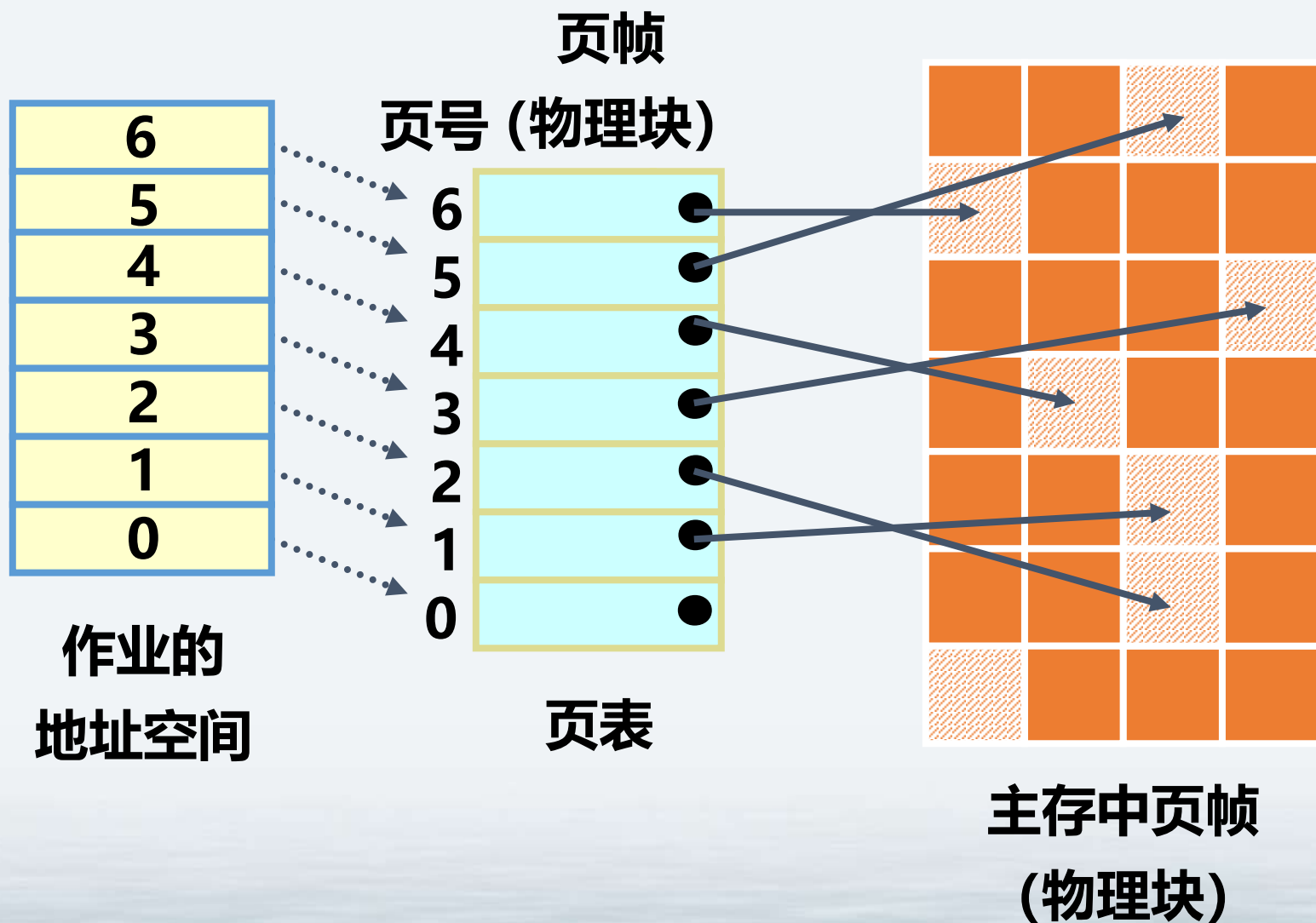
【例】

0		A0	A0	A0	A0	A0
1		A1	A1	A1	A1	A1
2		A2	A2	A2	A2	A2
3		A3	A3	A3	A3	A3
4			B0	B0		D0
5			B1	B1		D1
6			B2	B2		D2
7				C0	C0	C0
8				C1	C1	C1
9				C2	C2	C2
10				C3	C3	C3
11						D3
12						D4
13						
14						

3.3.2 页表

- 系统为每个进程建立一张页面映象表，即页表。
- 页表记录每一个进程的虚页号到物理内存中页帧号之间的映射关系，其作用是实现从页号到页帧号的地址映射。

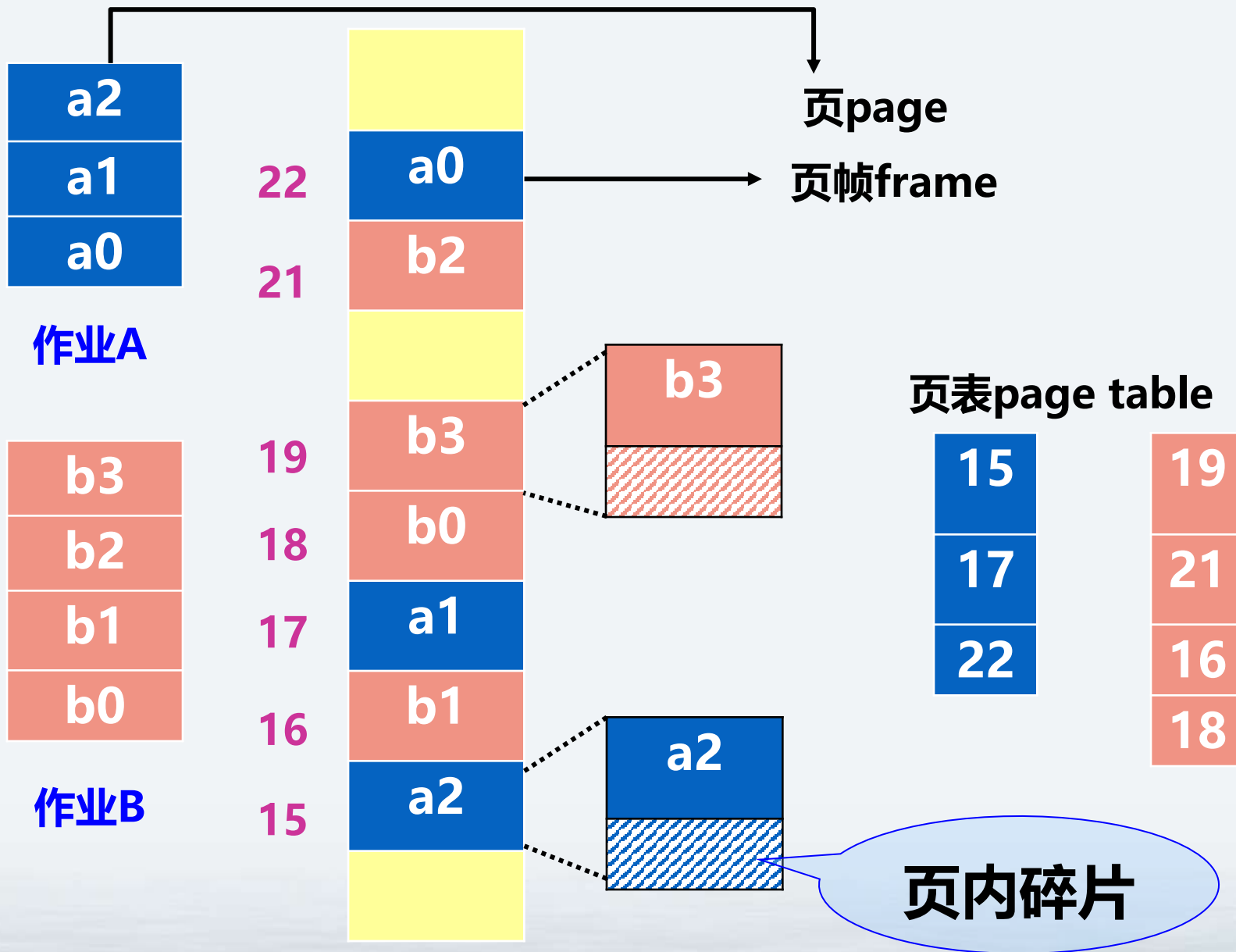
3.3.2 页表



3.3.2 页表

- 页表通常驻留在内存中；
- 系统设置一个页表寄存器PTR，存放当前执行进程的页表在内存的始址和页表的长度。
- 进程未执行时，页表的始址和长度存放在本进程的PCB中，当调度程序调度到某进程时，才将这两个数据装入PTR中。

内存



页面与页表

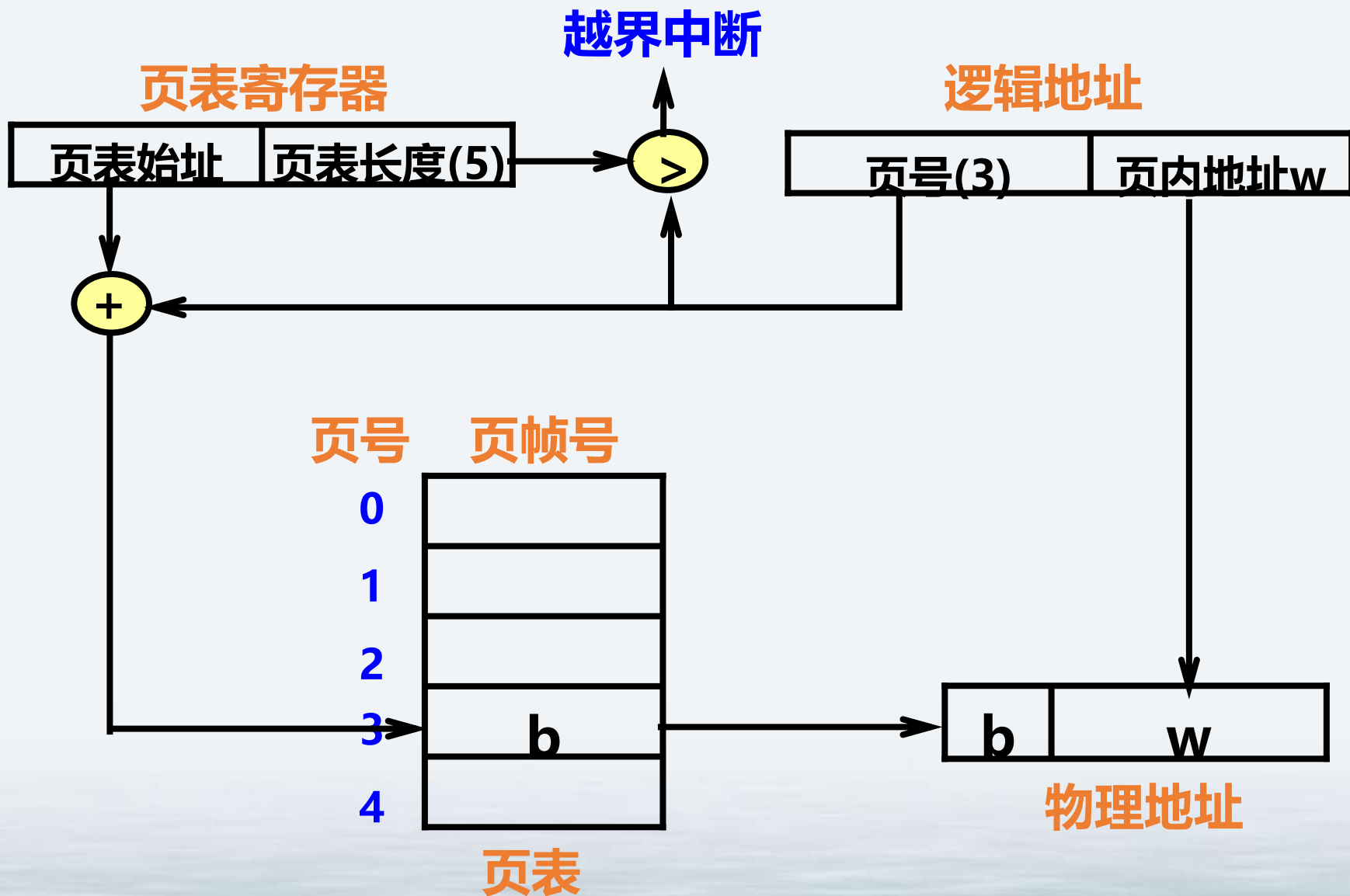
注意：页大小的选择

- 太大：页表长度小，但页内碎片大；
- 太小：内存碎片小，但页表过长；
- 页的大小是 2^k ，通常为512B-8KB。

地址变换过程

- 基本任务：逻辑地址→物理地址。
- 由于页面和物理页帧大小相同，因此页内地址和块内地址是一一对应的，无需变换。所以地址变换实际上只是将逻辑地址中的页号，转换为内存中的物理页帧号，这一任务是由页表来完成的。
- 即：页号→页表→物理页帧号b，与页内地址w合成，形成物理地址。

地址变换过程



页表

1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

110

在/不在
内存

物理地址

24580

MMU工作机制说明

页大小为4K

虚地址8196

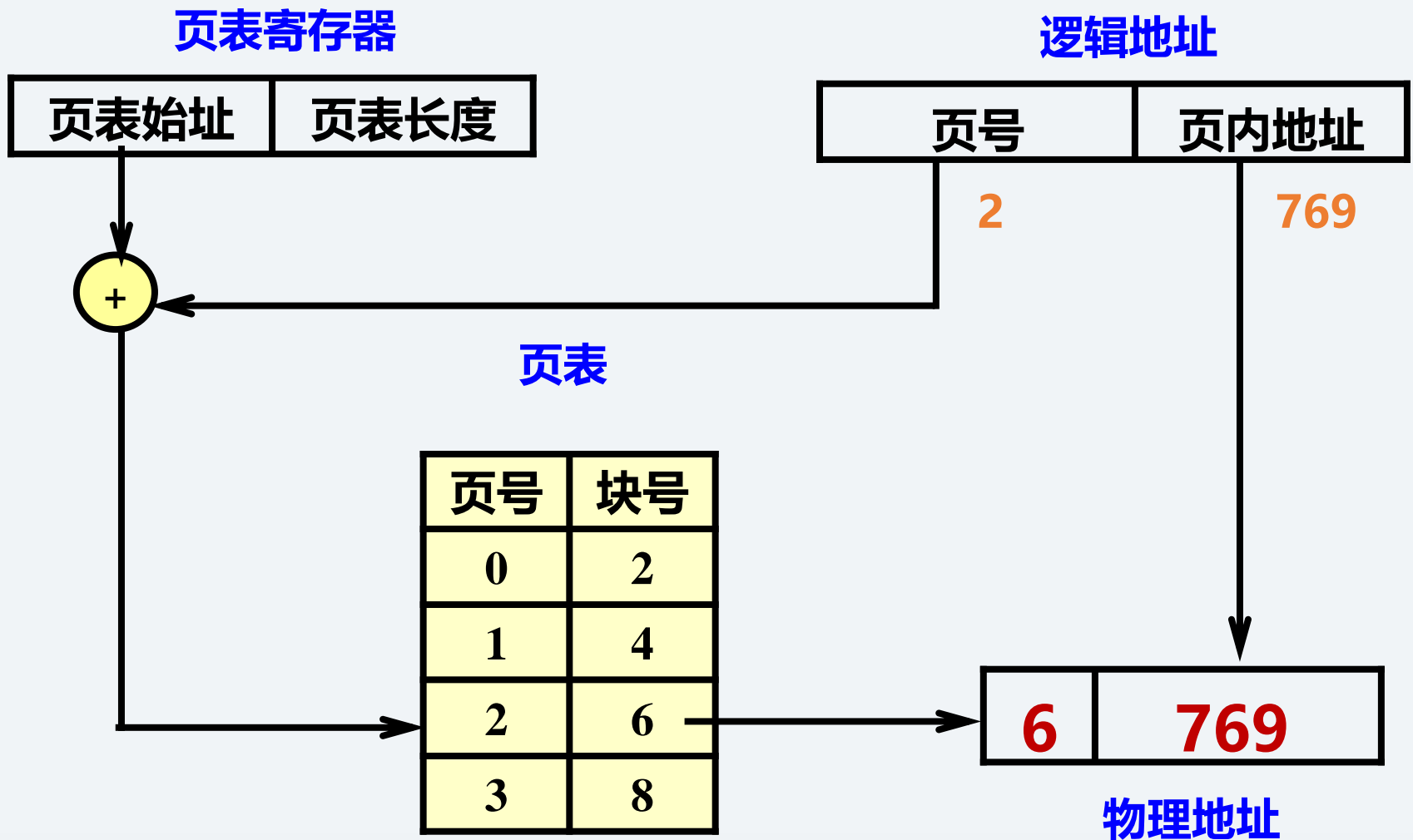
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

【例】

- 在采用页式存储管理的系统中，某作业J的逻辑地址空间为4页（每页2048字节），且已知该作业的页面映像表（即页表）如右图，试借助地址变换图（要求画出地址变换图）求出有效逻辑地址4865所对应的物理地址。

页号	块号
0	2
1	4
2	6
3	8

$$4865 = 2 * 2048 + 769$$



$$\text{物理地址: } 6 * 2048 + 769 = 13057$$

【思考题】

- 某虚拟存储器的用户空间共有32个页面，每页1KB，主存16KB。假定某时刻系统为该用户的第0、1、2、3页分别分配的物理页帧号为5、10、4、7，试将虚拟地址0A5C变换为物理地址。

【问题的解】

- 虚拟地址为32KB，则至少需要15位虚拟地址。首先，将给出的虚拟地址转化为二进制表示形式。

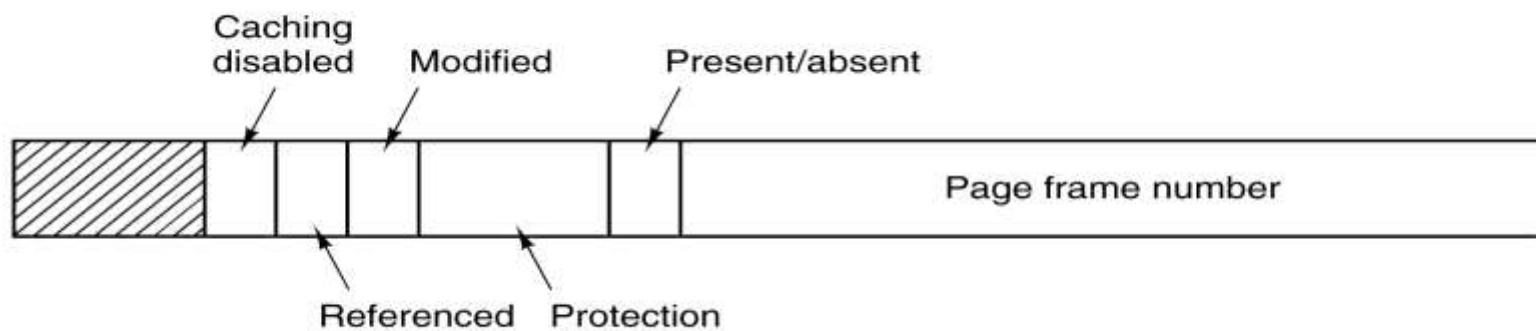


【问题的解】

- 虚拟页号(00010)为2，其对应的物理页帧号为4，由于主存为16KB，因此用4位表示页号，即0100，因此物理地址为物理页帧号合并上页内偏移，即为：

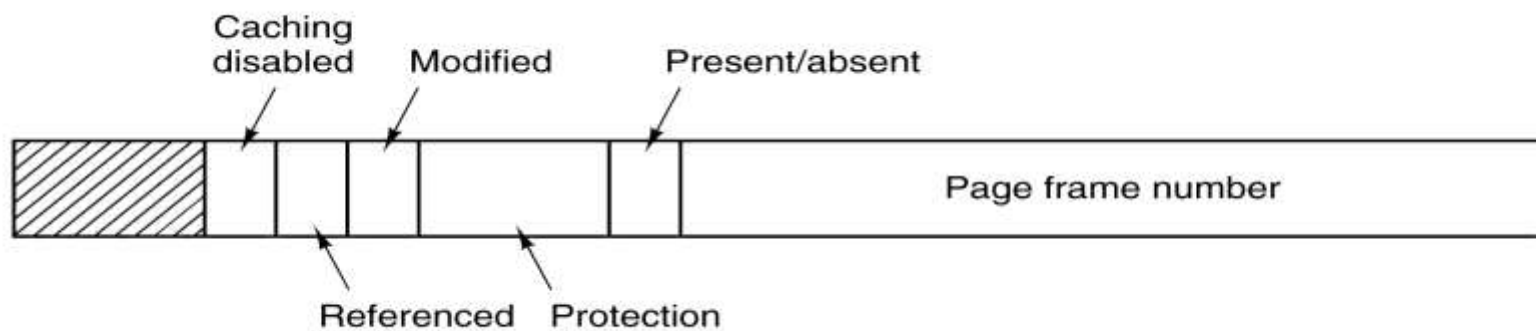


页表表项的结构



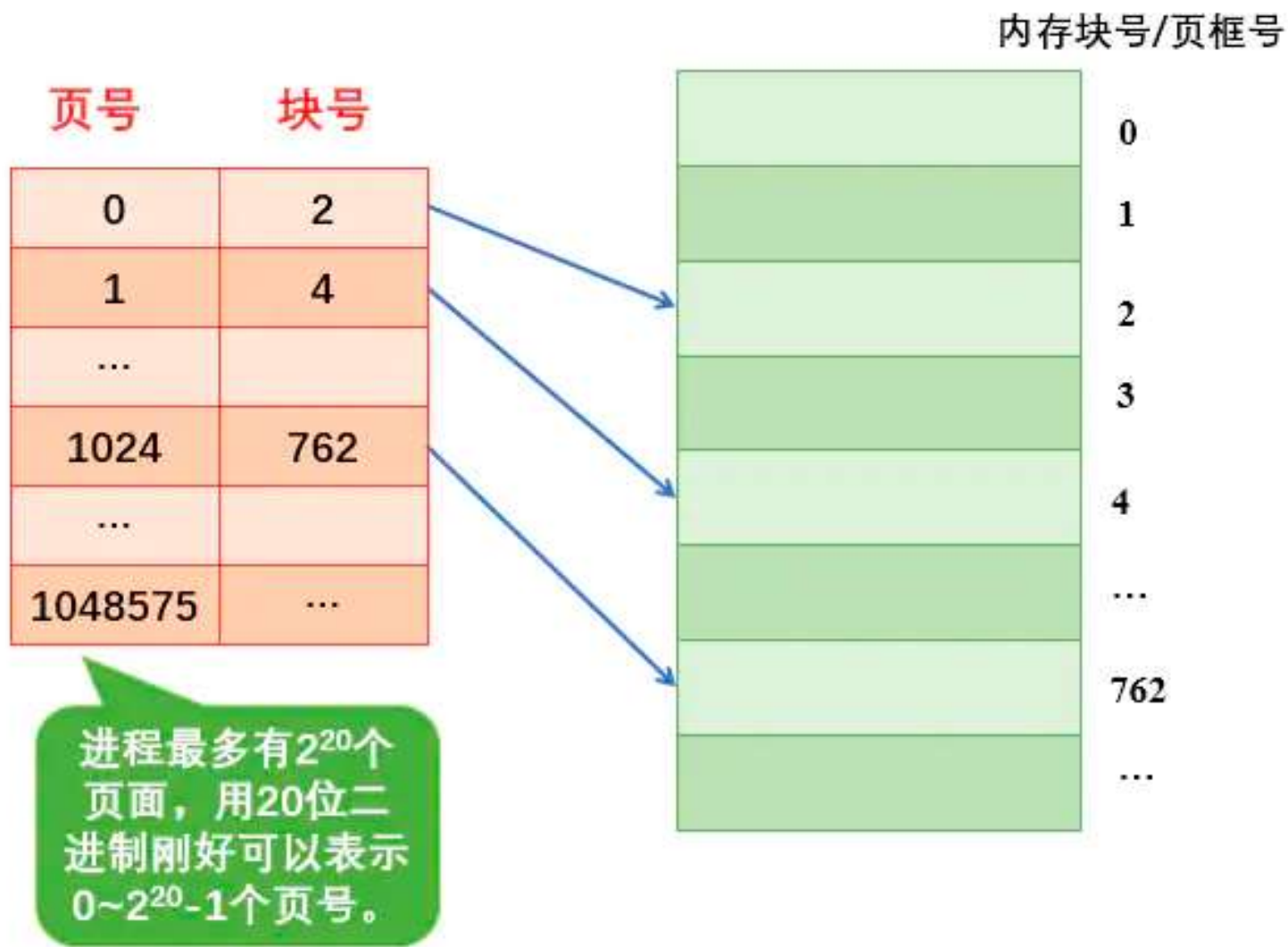
- 页帧号：虚拟页号在内存中的页编号
- 驻留位（在/不在位）：记录该逻辑页是否在内存中
- 保护位：读/写/执行，用来实现对内存的有效保护

页表表项的结构



- 修改位：记录该页内容是否被修改，用于保证数据的一致性
- 访问位：记录被访问的频率，用于缺页中断时对页面的选择替换
- 禁止高速缓存位：用内存映射I/O设备寄存器时，禁止高速缓存将保证数据一致性

两级页表和多级页表



单级页表存在的问

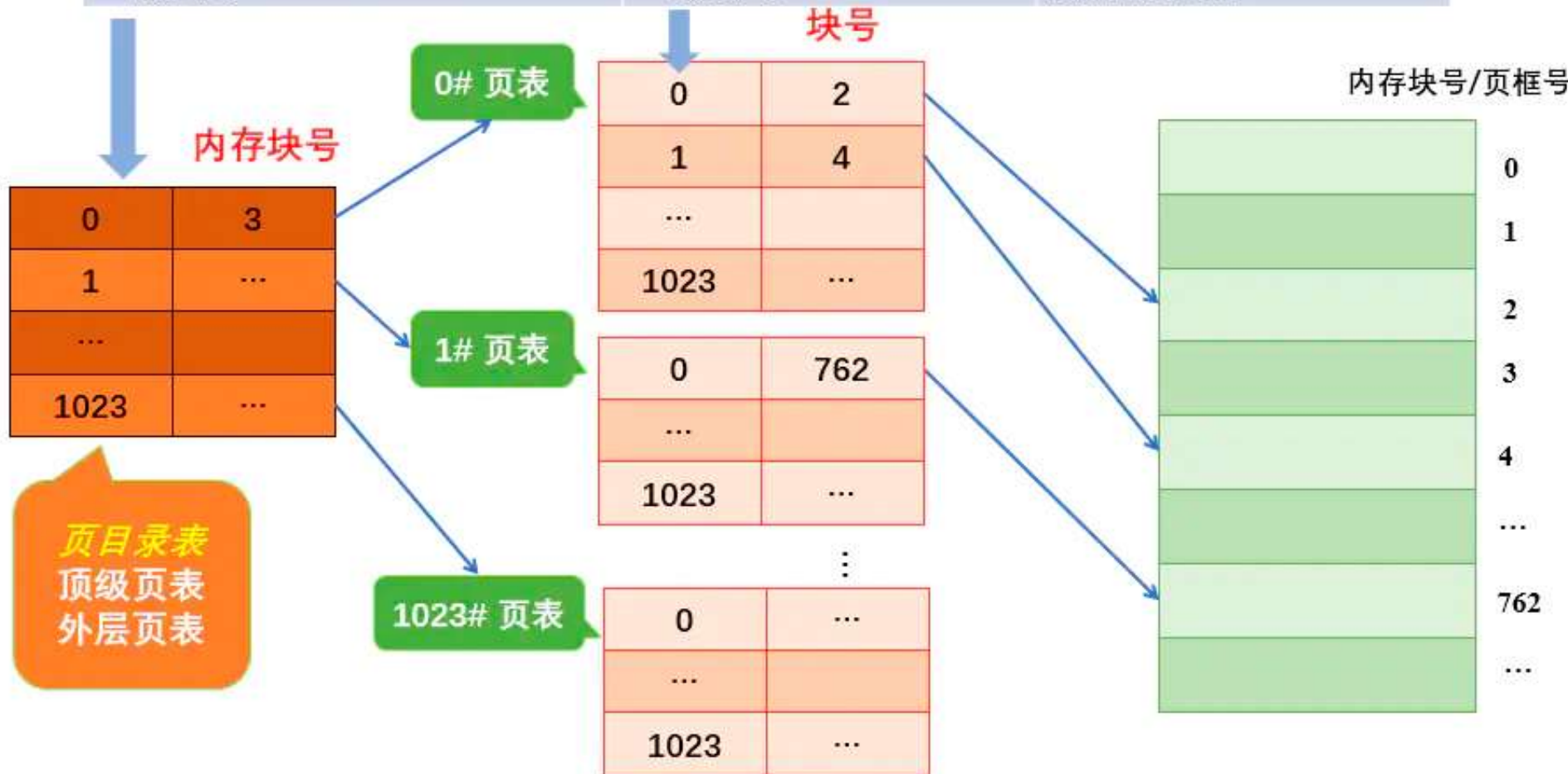
- 页表需要大量连续页面进行存储
- 在一段时间内，并非所有的页面都会被访问到，因此没有必要让整个页表常驻内存

两级页表和多级页表

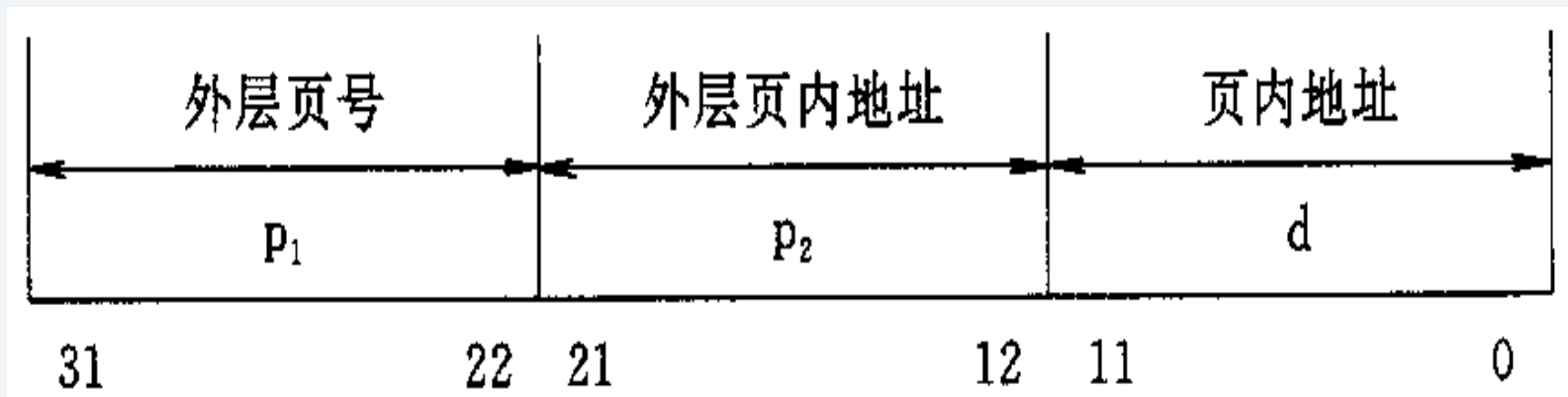
- 当页表项很多时，仅采用一级页表需要大片连续空间，可将页表也分页，并对页表所占的空间进行索引形成外层页表。由此构成两级页表。
- 更进一步可形成多级页表。

10位正好可以表示页号从0~1023

31	22	21	12	11	0
一级页号			二级页号			页内偏移量		



两级页表 (Two-Level Page Table)

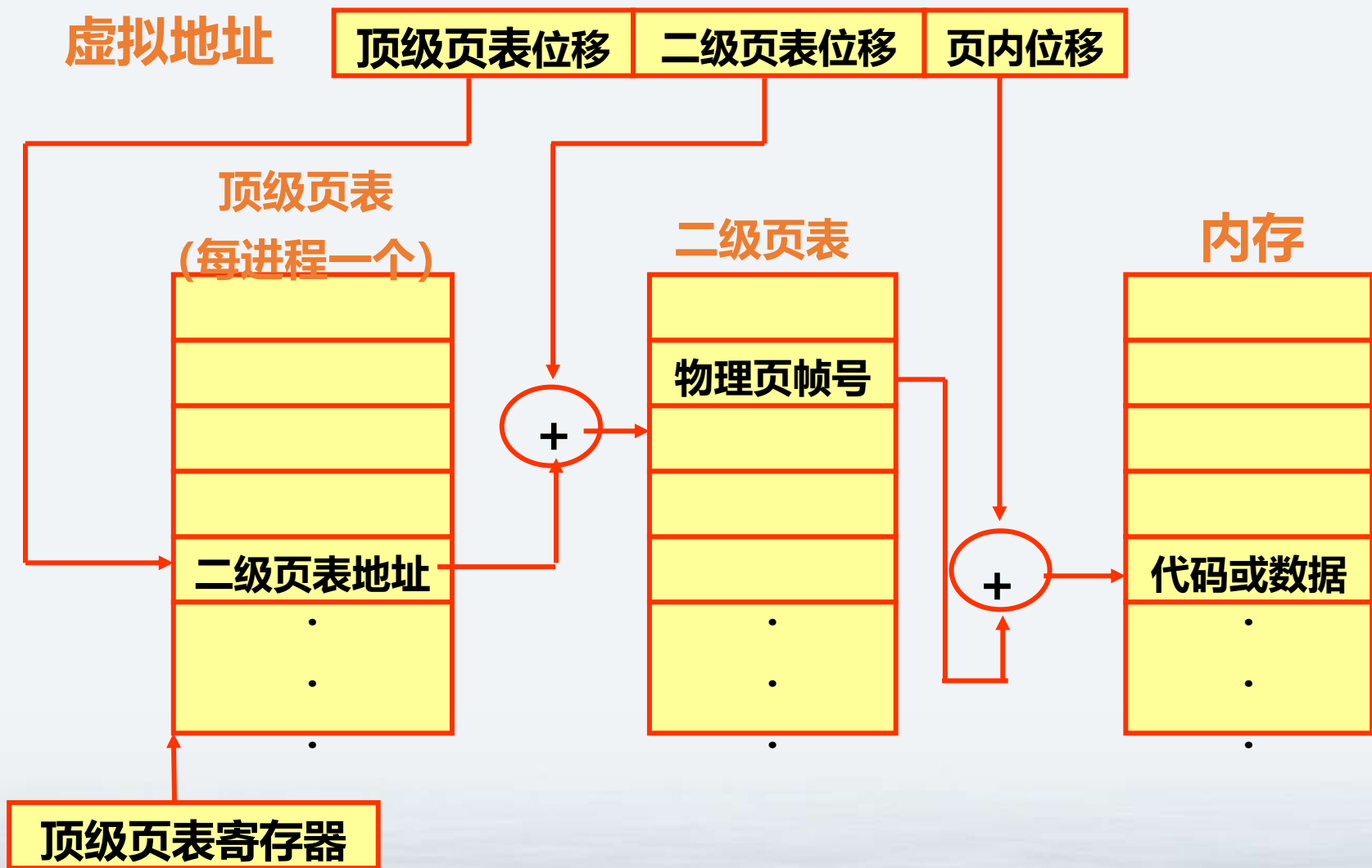


外层页表有 2^{10} 个页表项;

最多允许有 2^{10} 个页表分页;

页面大小为: $2^{12} = 4\text{KB}$ 。

具有两级页表的地址变换



两级页表的地址变换

- ① 按照地址结构将逻辑地址拆成三个部分。
- ② 从PTR中读取顶级页表起始地址，再根据一级页号查顶级页表，找到下一级页表在内存中存放位置。
- ③ 根据二级页号查表，找到最终想要访问的物理页帧号。
- ④ 结合页内偏移量得到物理地址。

31	22	21	12	11	0
一级页号			二级页号			页内偏移量		

逻辑地址 (0000000000,0000000001,1111111111)

内存块号

0	3
1	...
...	
1023	...

块号

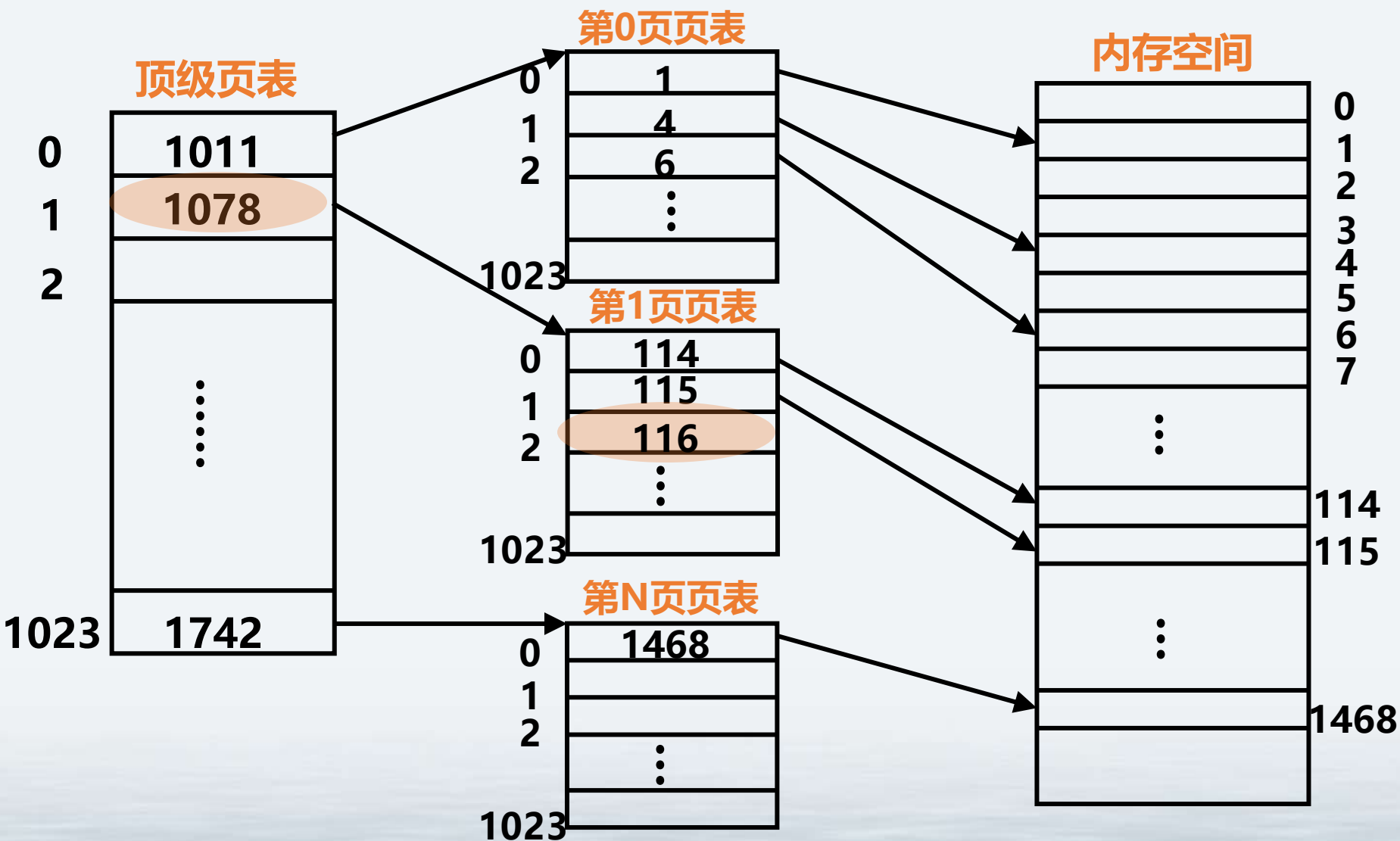
0	2
1	4
...	
1023	...

内存块号/页框号

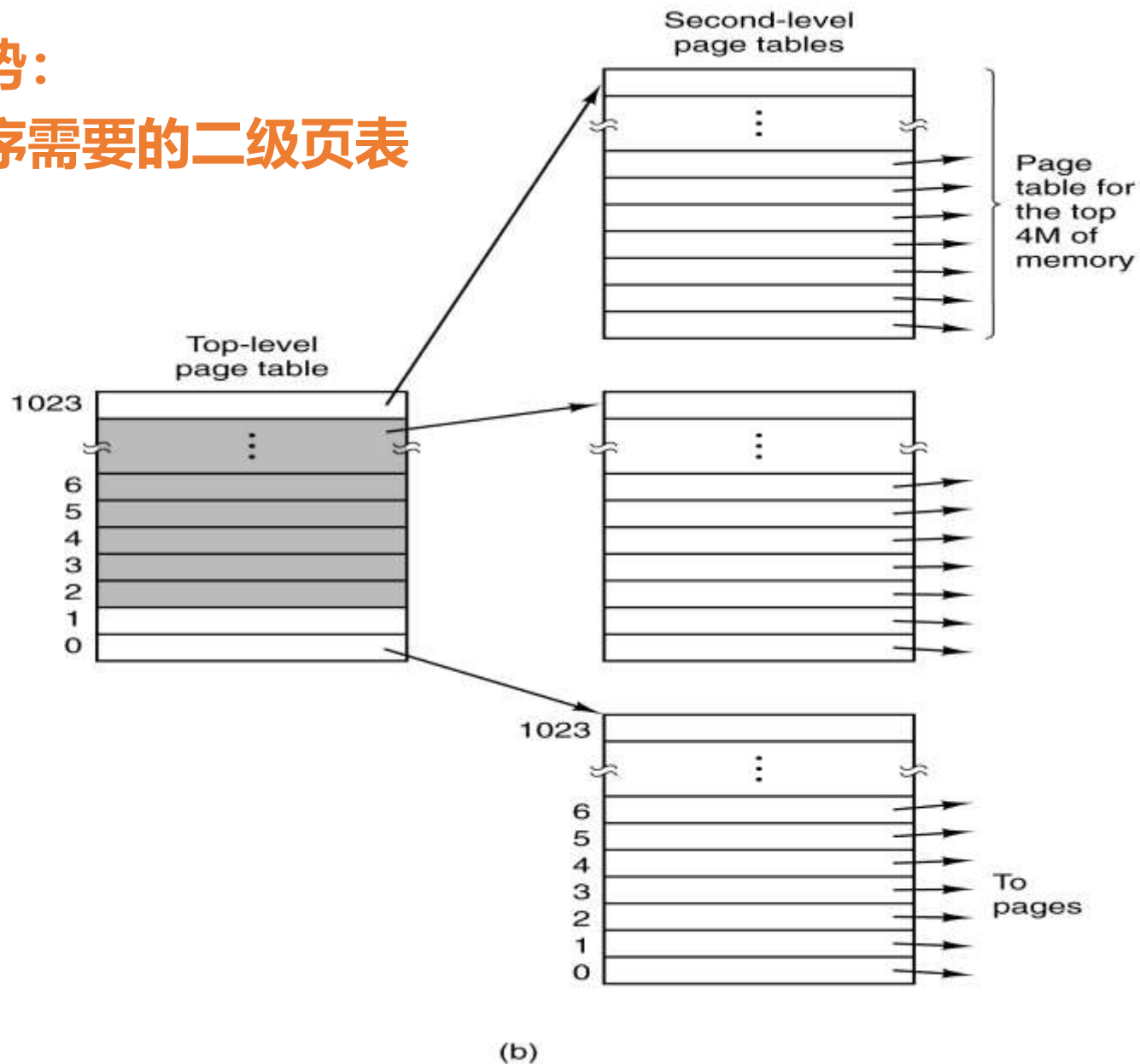
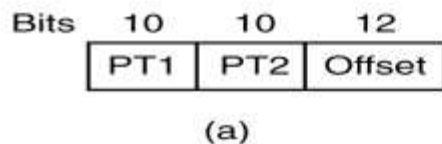
	0
	1
	2
0#页表存放的位置	3
	4
	...
	762
	...

先访问页目录表的0#页表，得到0#页表在内存块中的位置是**3**号内存块，这样就可以得到0#页表的起始地址，然后在访问0#页表的1号页号，得到内存块号为**4**号内存块。
所以，最终访问的地址为： $4 * 4096 + 2^{12} - 1 = \mathbf{20479}$

求虚拟地址0x00402004对应的物理地址



多级页表的优势： 只需要保存程序需要的二级页表



思考题

- 某系统按字节编址，采用40位逻辑地址，页面大小为4KB，页表项大小为4B，假设采用纯页式存储，则要采用（）级页表，页内偏移量为（）位？

逻辑地址:

页号 28位

页内偏移量 12位

逻辑地址:

一级页号 8位

二级页号 10位

三级页号 10位

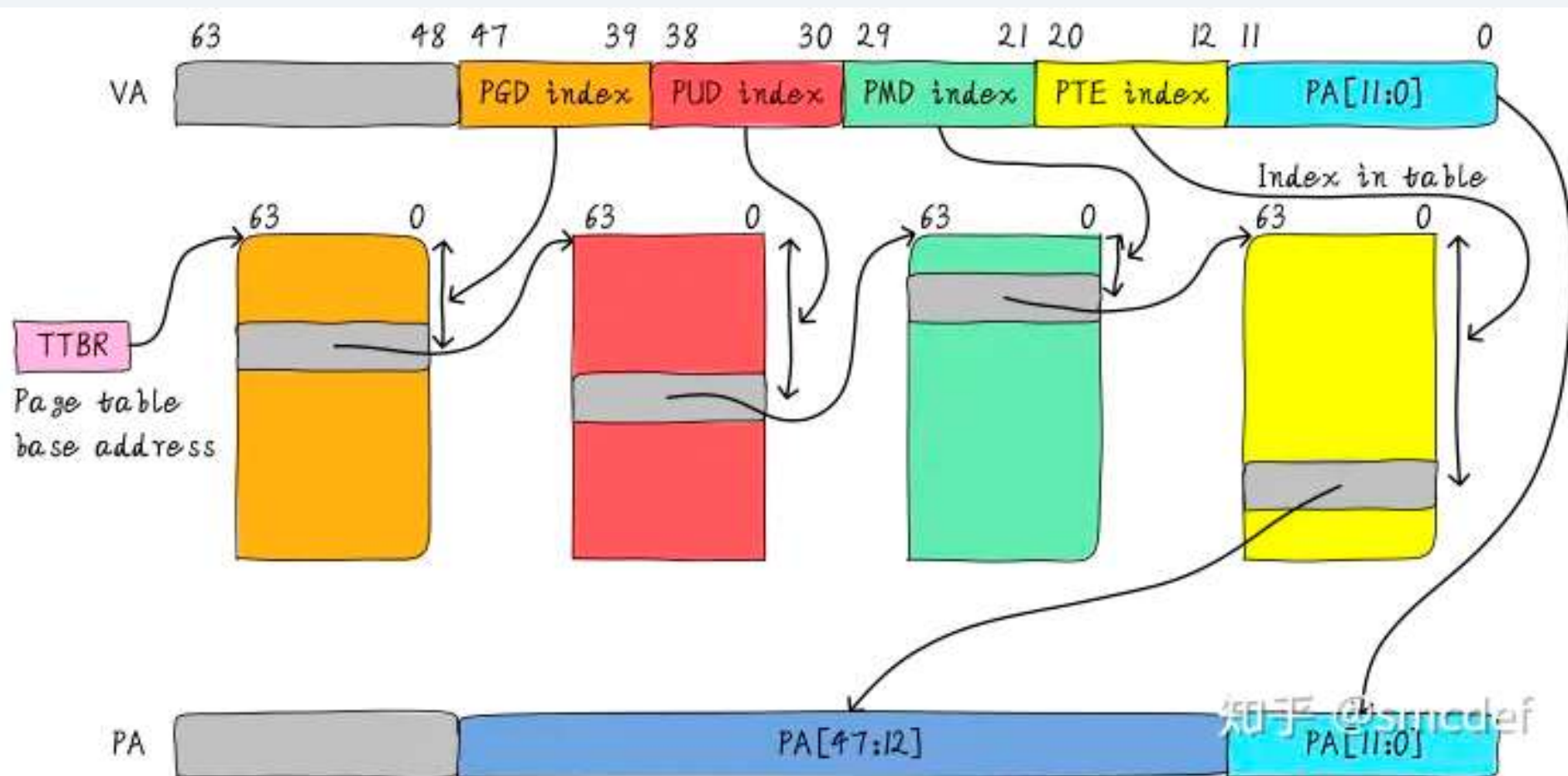
页内偏移量 12位

- 页面大小 = 4KB, 按字节编址, 因此页内偏移量为12位。
- 页号 = $40 - 12 = 28$ 位。
- 页面大小 = 4KB, 页表项大小 = 4B, 则每个页面可存放1024个页表项。
- 因此各级页表最多包含1024个页表项, 需要10个二进制位才能映射到1024个页表项, 因此每级页表对应的页号应为10位二进制。共28位的页号至少要分为3级。

3.3.3 转换检测缓冲区

Qestion:

- 为了取出一个数据，系统需要访问内存几次？



3.3.3 转换检测缓冲区

- 页地址变换过程中，页表是放在内存中的，因此CPU在每存取一个数据时，都要两次访问内存。第一次是访问内存中的页表，得到物理地址。第二次访问内存时，才得到所需的数据。
- 采用这种方式，运行速度将下降一半。两级页表地址变换需三次访问内存，运行速度将下降 $\frac{2}{3}$ 。

3.3.3 转换检测缓冲区

- 为了提高存取速度，通常设置一个具有并行查询能力的特殊高速缓冲寄存器，存放当前进程最常用的页号及对应物理页帧号。这种设备称为转换检测缓冲区（TLB: Translation Lookaside Buffer）。

3.3.3 转换检测缓冲区

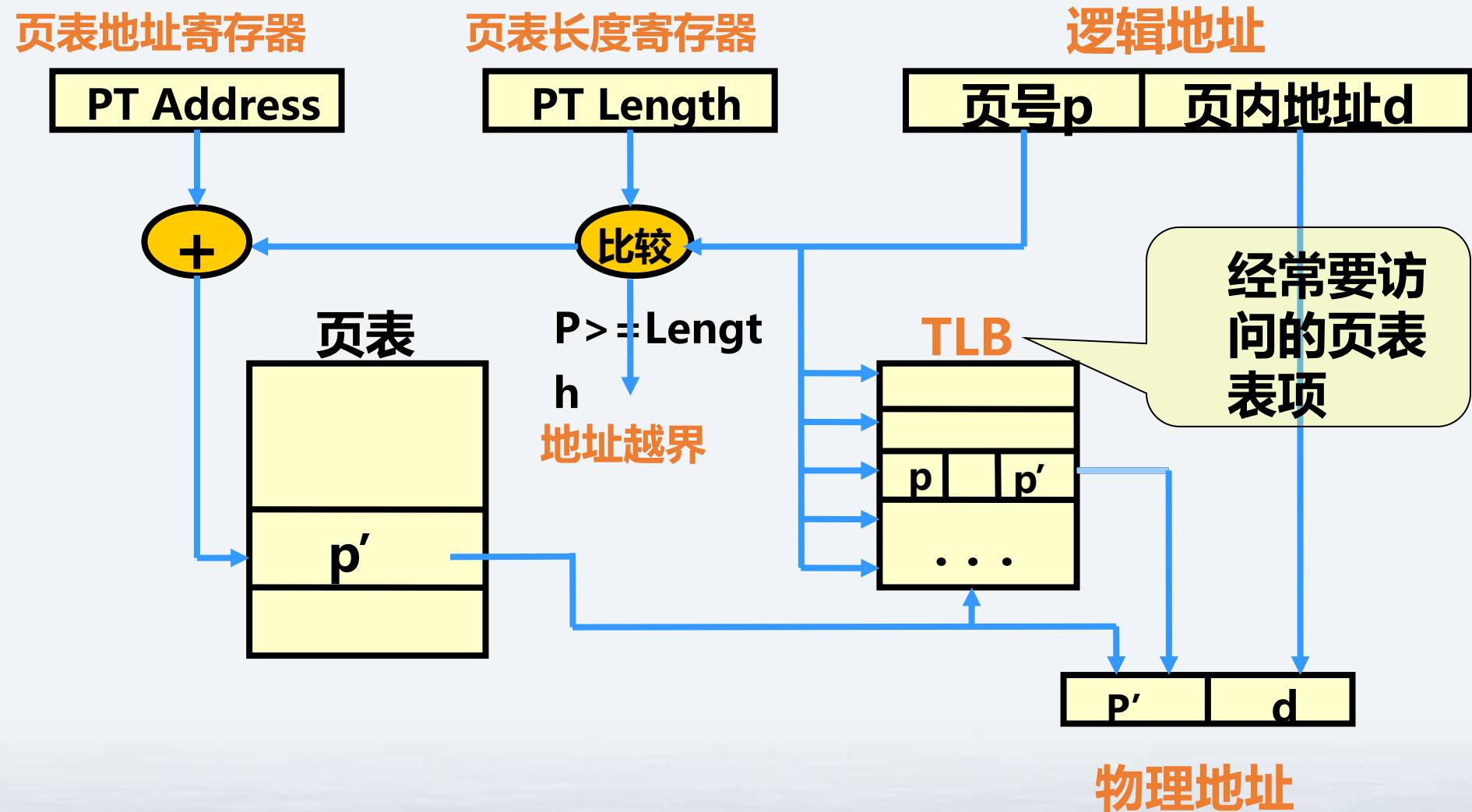
- 硬件加速机制TLB：
 - 关联存储器原理：将页号与TLB中的各行同时比较，实现快速查找
 - TLB位于MMU中，保存着最常被访问的页表项
 - 程序运行过程中直接使用TLB，当TLB失配时从页表中调入对应项
 - TLB机制原理：局部性原理 + 硬件加速

TLB结构组成示例

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

思考：为什么TLB中要保存虚拟页号而页表中不要？

硬件加速机制的使用 (TLB)



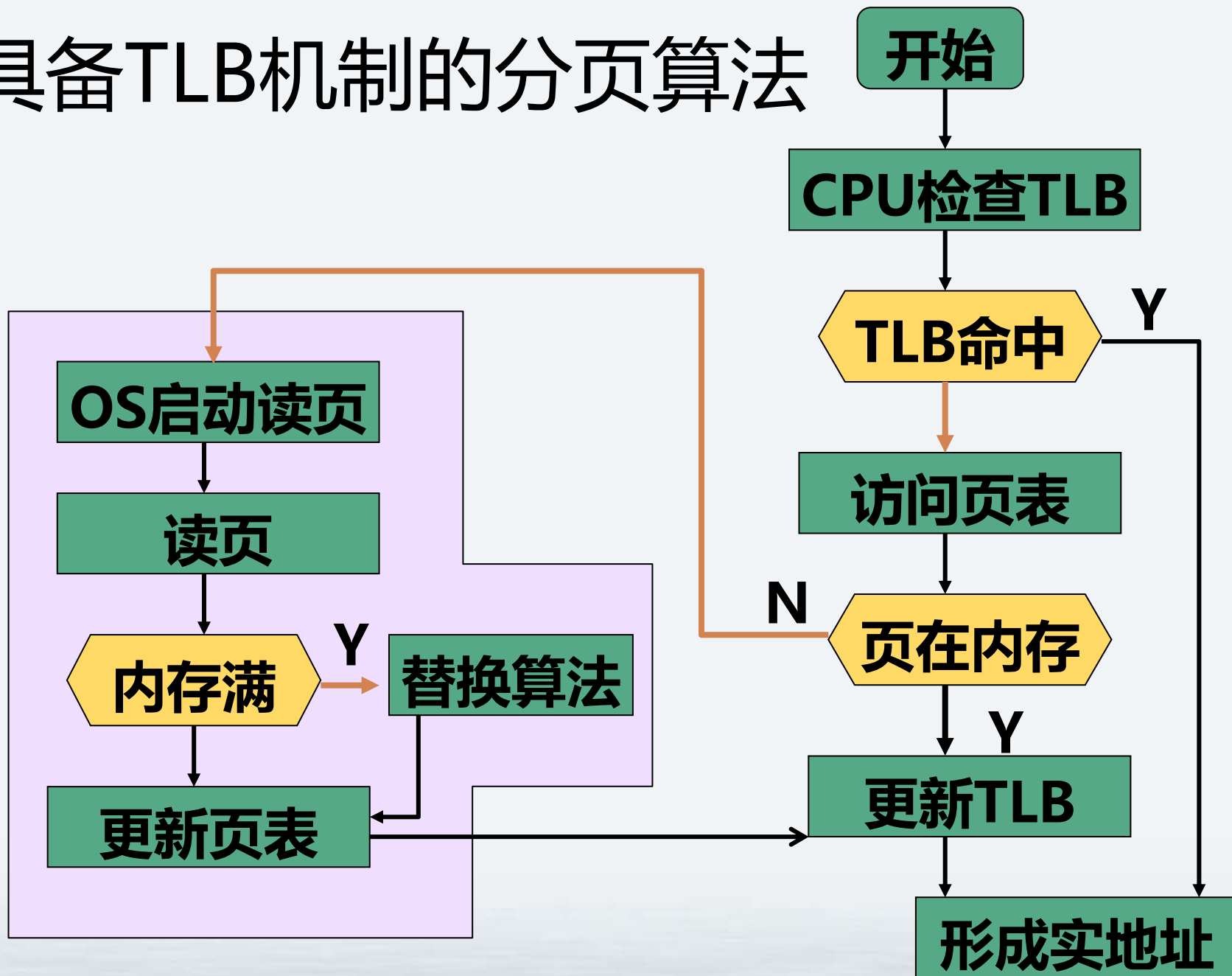
分页存储管理机制的工作流程

- 程序开始运行阶段
 - 创建进程结构，内存中并无任何页面
 - CPU执行指令，经过短暂的“颠簸”，形成“CPU - TLB - 页表 - 外存”

分页存储管理机制的工作流程

- 程序运行过程中
 - PC获取下一条指令时首先访问TLB，如果命中则直接获得内存地址
 - 如果TLB未命中，则访问页表，将页表中该项替换进入TLB
 - 如果内存中的页表也未命中，则发生缺页中断，将指令从外存中调入，替换某个内存页面

具备TLB机制的分页算法



3.3.4 倒排页表

- 若虚拟地址为64位，采用一级页表，虚拟地址空间为 2^{64}B ，页面大小为4KB，则需 2^{52} 个表项，若每个表项需8字节.....



传统页表

4K页面的传统页表

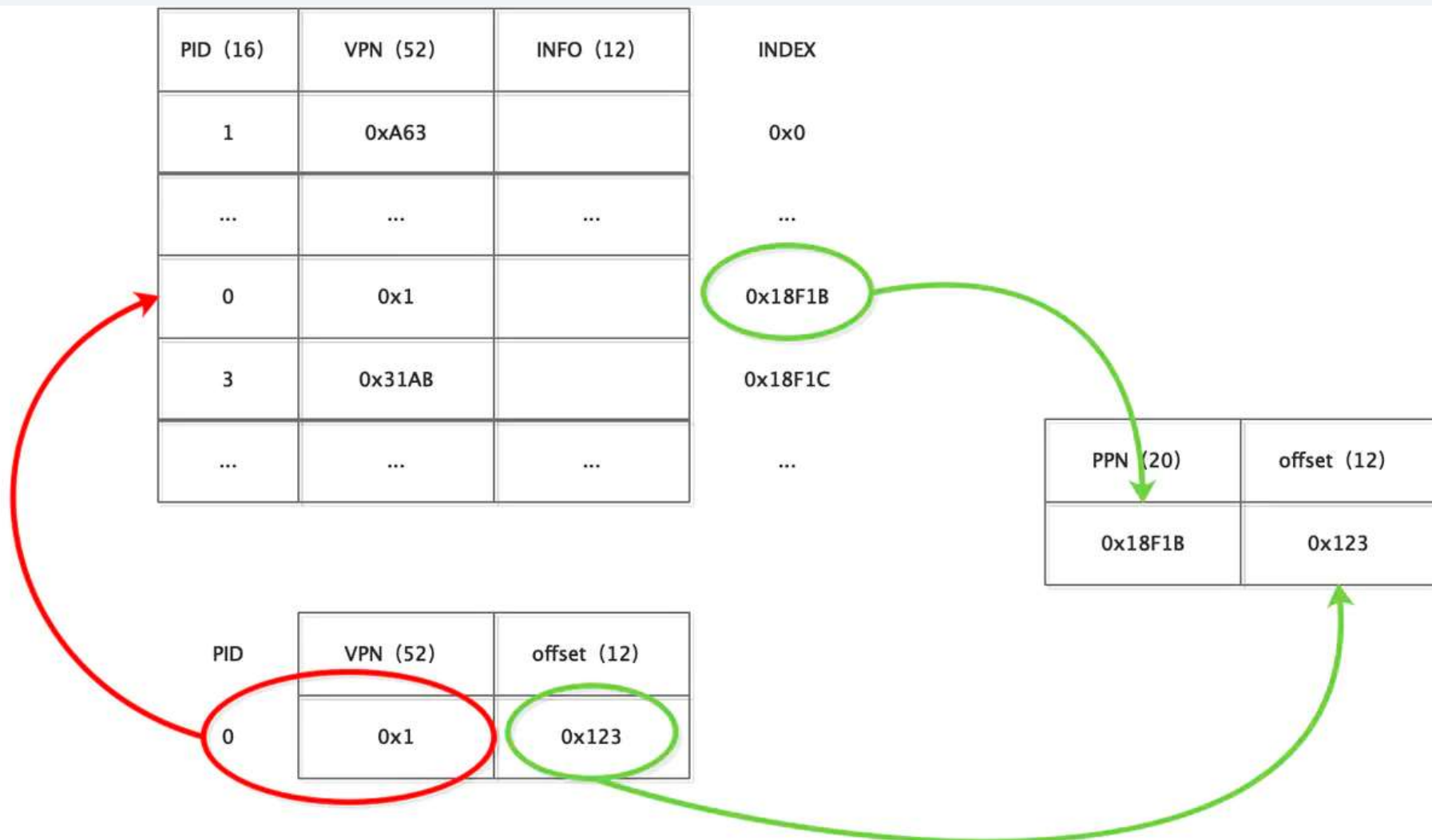


3.3.4 倒排页表

- 基本思想：在实际内存中每一个页帧有一个表项，而不是每一个虚拟地址空间有一个表项。

3.3.4 倒排页表

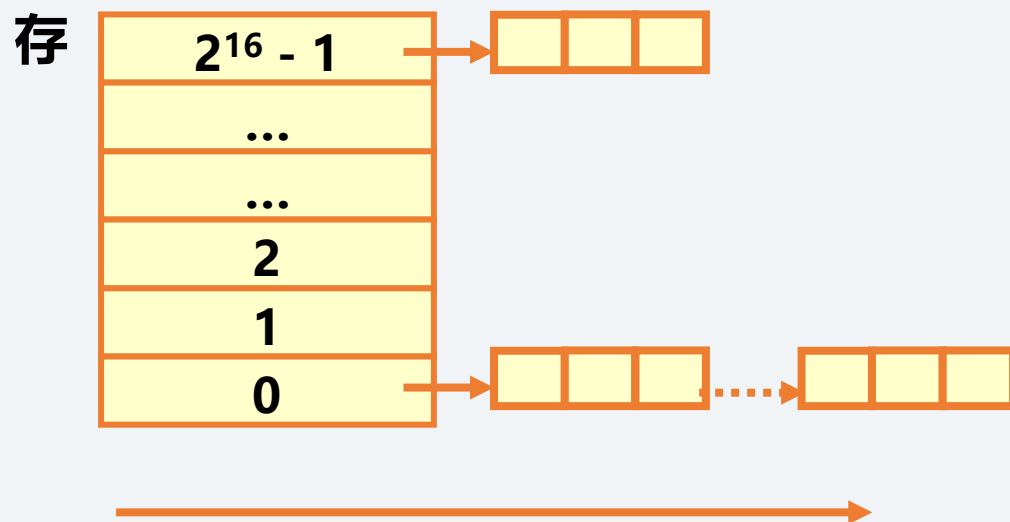
需要搜索整个倒排页表来寻找某一表项



3.3.4 倒排页表



4K页面的256M内存



利用虚拟地址来散列

使用散列技术提高映射速度

4.3.4 倒排页表

倒排页表的特点分析

- 当虚拟地址空间远大于物理地址空间时适用
- 通过散列技术提高地址映射速度
- 64位乃至更高位计算环境的合理选择

虚拟页式存储技术

- Q: 当内存中没有空闲页面时, 如果还要调入一个新页, 如何处理?

3.4 页面置换算法

- 系统在进程开始运行之前，不是装入全部页面，而是装入一个或几个页面，之后根据进程运行的需要，动态装入其它页面。
- 当主存中没有空闲的物理页时，为了要接受一个新页，需要把老的一页淘汰出去。

3.4 页面置换算法

Q：根据什么策略选择欲淘汰的页面？

- 出发点：希望把未来不再使用的或者短时间内较少使用的页面调出。

3.4 页面置换算法

几种常用置换算法：

- 最佳置换算法(OPT, Optimal)
- 最近未被使用置换算法(NRU)
- 先进先出置换算法(FIFO)
- Clock置换算法
- 最近最久未使用置换算法(LRU, Least Recently Used)
- 最少使用置换算法(LFU, Least Frequently Used)

3.4.1 最佳(Optimal)置换算法

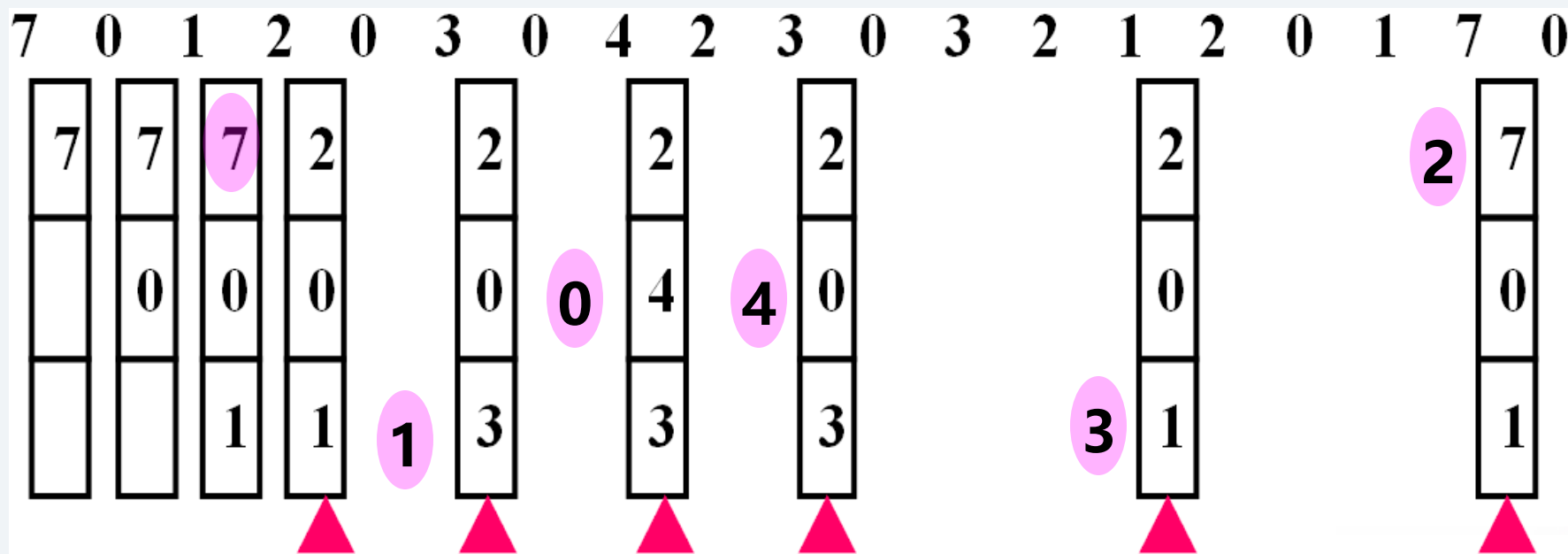
- Belady于1966年提出的一种理论上的算法。
- 选择“未来不再使用的”或“在最长时间不再被访问的”页面被置换。
- 通常可以获得最低的缺页率。
- 这是一种理想情况，是实际执行中无法预知的，因而不能实现。可用作性能评价的依据。

3.4.1 最佳(Optimal)置换算法

- 例：假定系统为某进程分配了三个物理页帧， 并考虑有以下的页面号引用串：7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1。

3.4.1 最佳(Optimal)置换算法

利用最佳页面置换算法时的置换图：



整个过程缺页中断发生了9次，页面置换发生了6次。

缺页率 = $9 / 20 = 45\%$ 。

3.4.2 最近未使用页面置换算法

- 页表项中设置了“访问位”和“修改位”，初始值均为0，可用R、M标识
- 当某个页面被读取（Read）时，“访问位”设置为1
- 当某个页面被修改（Modify）时，“修改位”设置为1
- “访问位”定期清零(如在每一个时钟中断时)
- 缺页中断发生时，对所有页面的R、M值进行分析，从中选择最近未使用页面进行替换

3.4.2 最近未使用页面置换算法

- 最近未使用页面的分类（依据R、M值）
 - 第0类（00）：没有被访问，没有被修改
 - 第1类（01）：没有被访问，已经被修改
 - 第2类（10）：已经被访问，没有被修改
 - 第3类（11）：已经被访问，已经被修改

3.4.2 最近未使用页面置换算法

NRU算法的核心理念

- 访问位要比修改位重要：替换一个被修改的页面要比替换一个被访问的页面更容易保证性能。NRU算法随机地从编号最小的非空类中挑选一个页面淘汰。

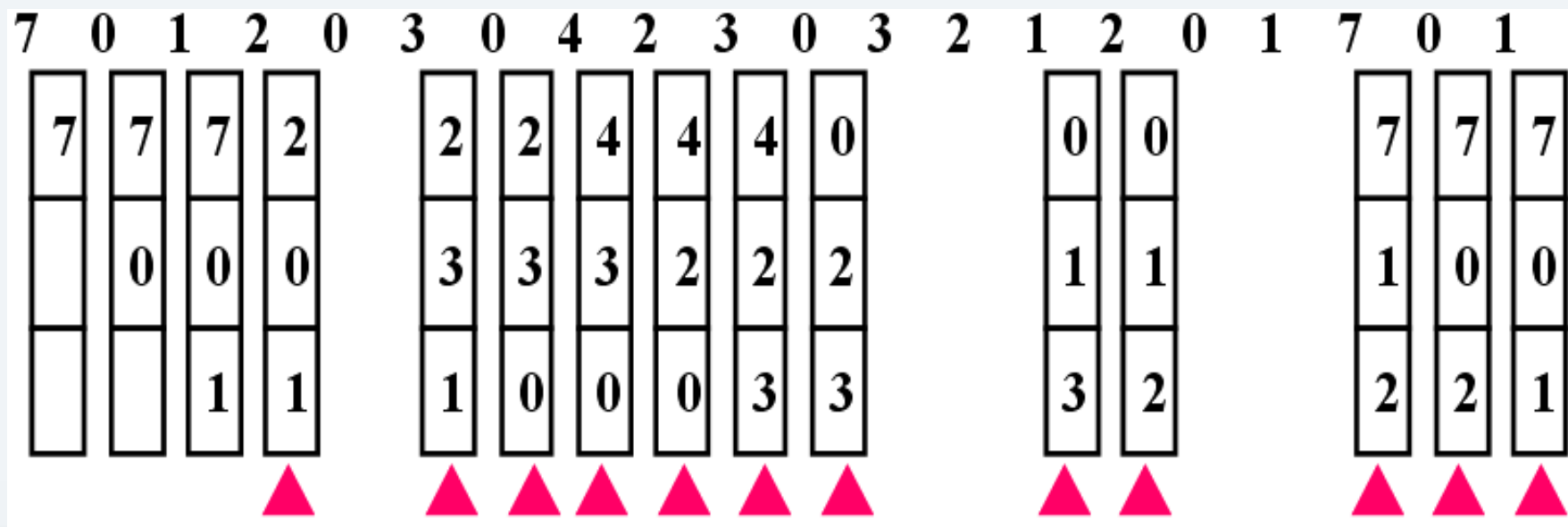
3.4.3 先进先出页面置换算法

- 基本思想：总是先淘汰那些驻留在内存时间最长的页面，即先进入内存的页面先被置换掉。
- 操作系统维护着一个页面链表，所有页面按照进入内存的时间先后顺序排列。
- 当需要替换页面时，淘汰表头的页面并把新调入的页面加到表尾。

3.4.3 先进先出页面置换算法

例：（同前例）

利用FIFO置换算法时的置换图：



共发生12次页面置换

3.4.3 先进先出页面置换算法

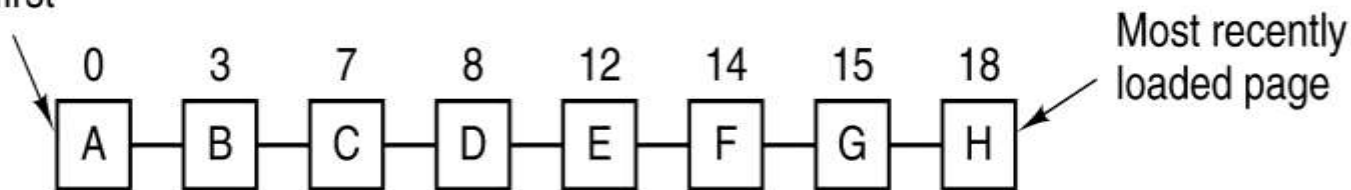
- 易于实现，但是容易将被频繁访问的“老”页面替换出去。进入内存时间的长短与被使用的频率之间并无直接关联。

3.4.4 第二次机会页面置换算法

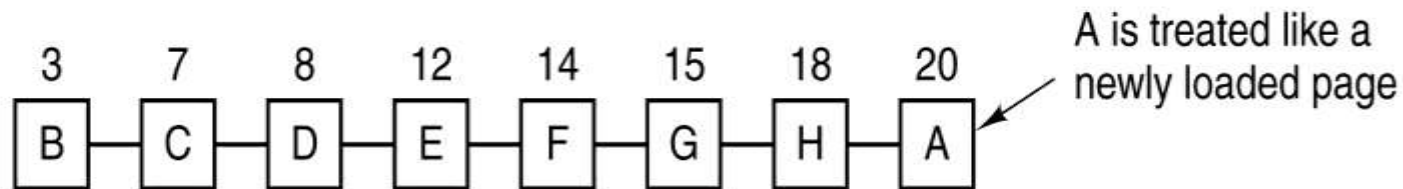
- 淘汰不但“老”而且（最近）“没用”的页面。
- 原理：
 - 用链表来表示各页的建立时间先后，新来的到表尾，表头就是最“老”的（同FIFO）。
 - 选择淘汰页面时，若表头页面的R位（访问位）是0，则淘汰之，否则将其R位设为0，并把它放到表尾，然后继续从表头搜索（页面装入时设 $R=1$ ）。

3.4.4 第二次机会页面置换算法

Page loaded first



(a)



(b)

3.4.4 第二次机会页面置换算法

- 对FIFO的修改，R位为0则替换，R位为1则置零并将其放在链表尾部。
- 寻找一个在上一个时钟间隔以来没有被访问的“老”页面。如果所有的页面都被访问过了，该算法就简化为纯粹的FIFO算法。

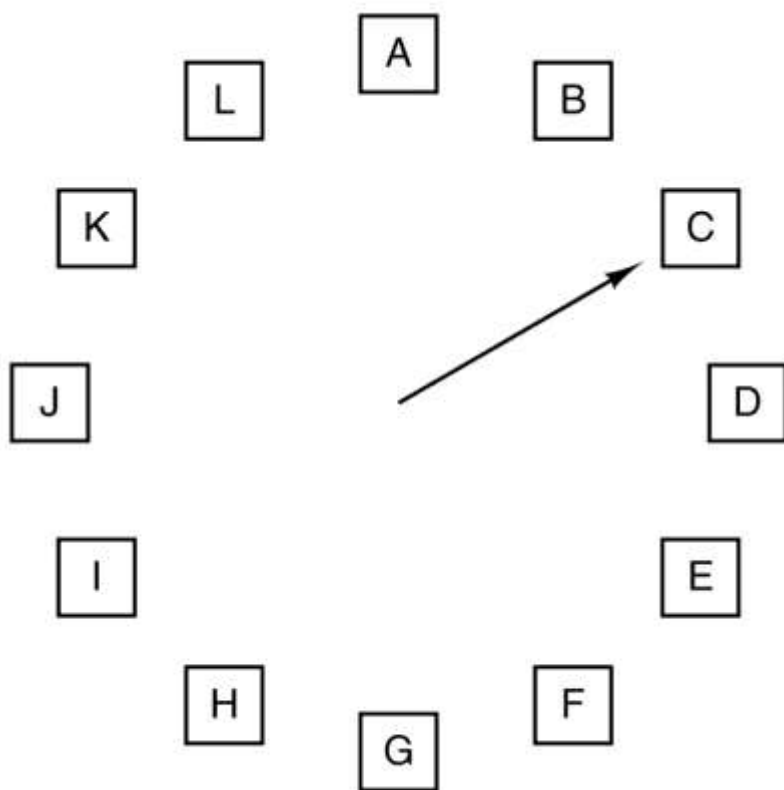
3.4.4 第二次机会页面置换算法

- 第二次机会页面置换算法较FIFO合理，但需经常在链表中移动页面，降低了效率。

3.4.5 时钟页面置换算法

- 对“第二次机会”算法的优化处理，实现过程类似“钟盘”。
- 将页面按FIFO原则组织成一个环形链表，设一个指针指向最老的页面。

3.4.5 时钟页面置换算法

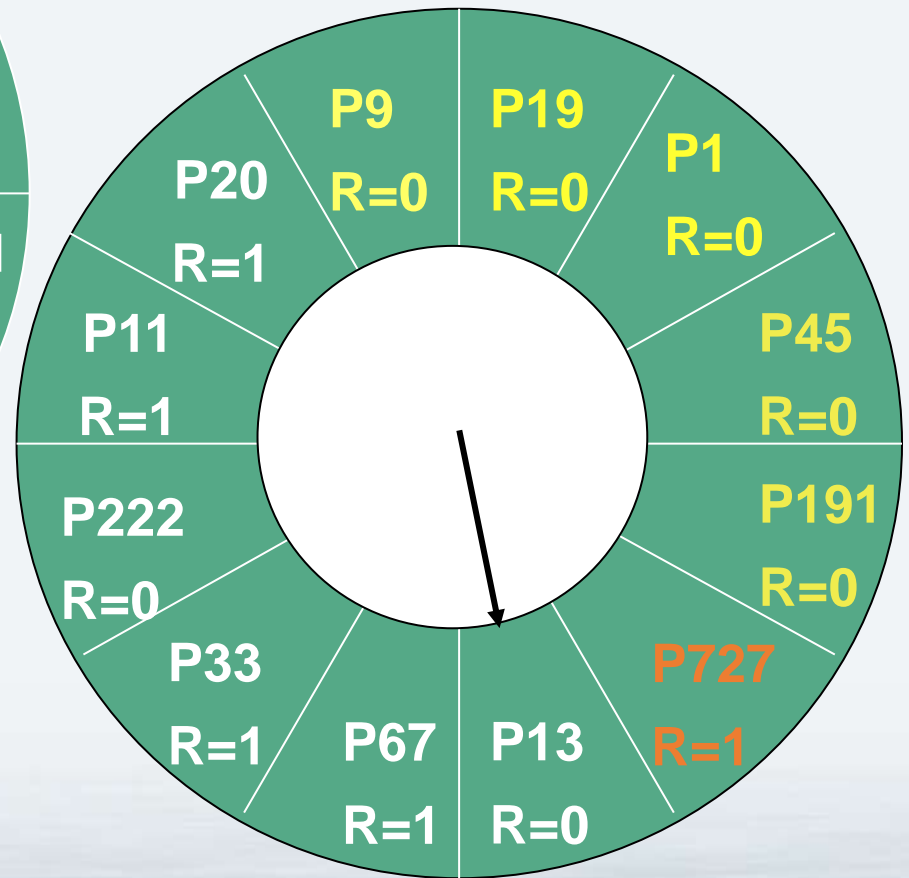
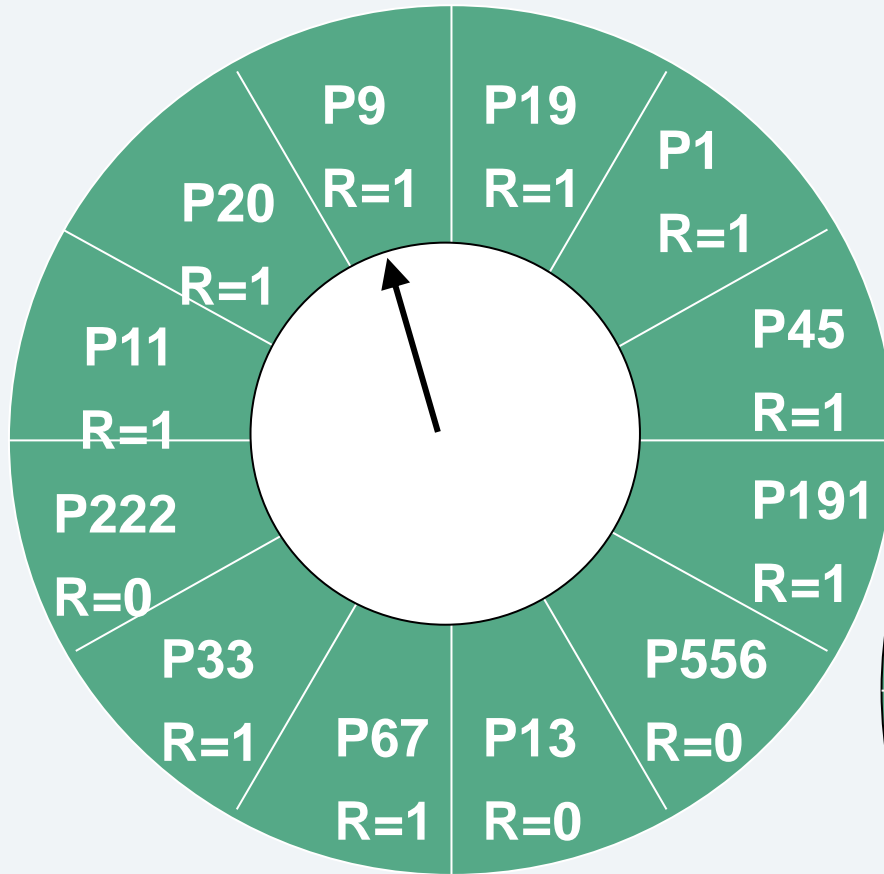


When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

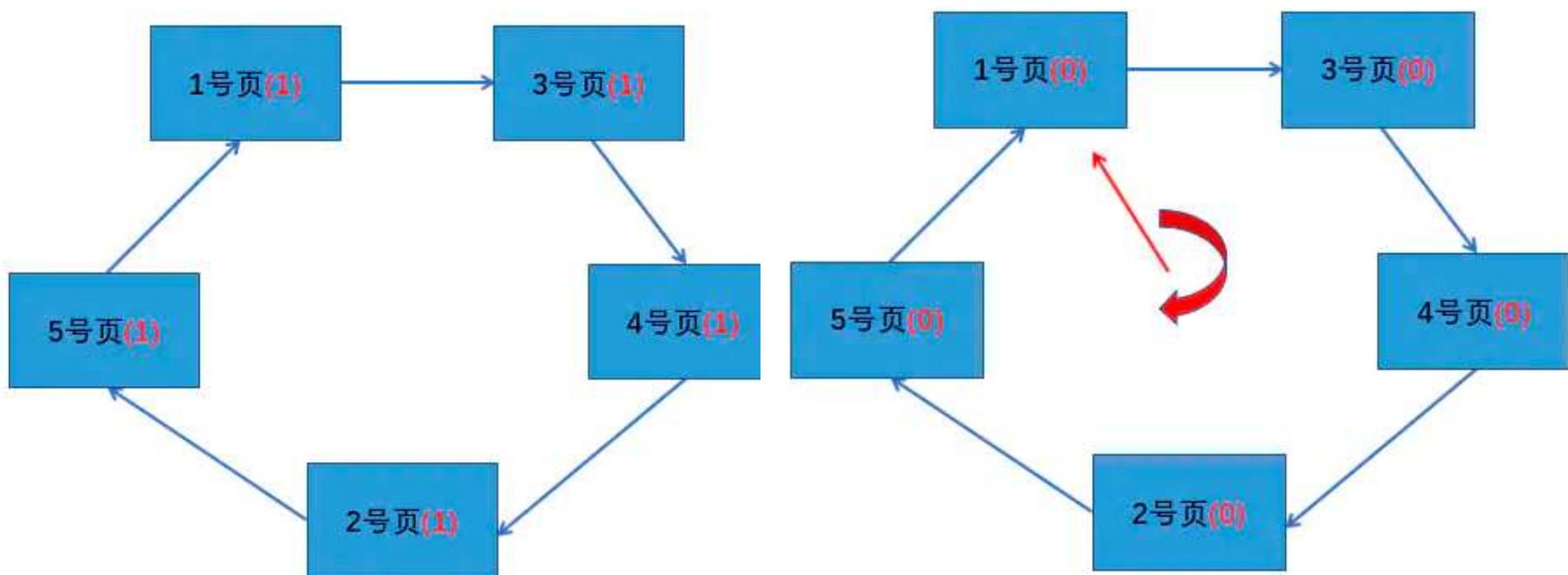
R = 0: Evict the page

R = 1: Clear R and advance hand

Clock策略



所有的页面都被访问



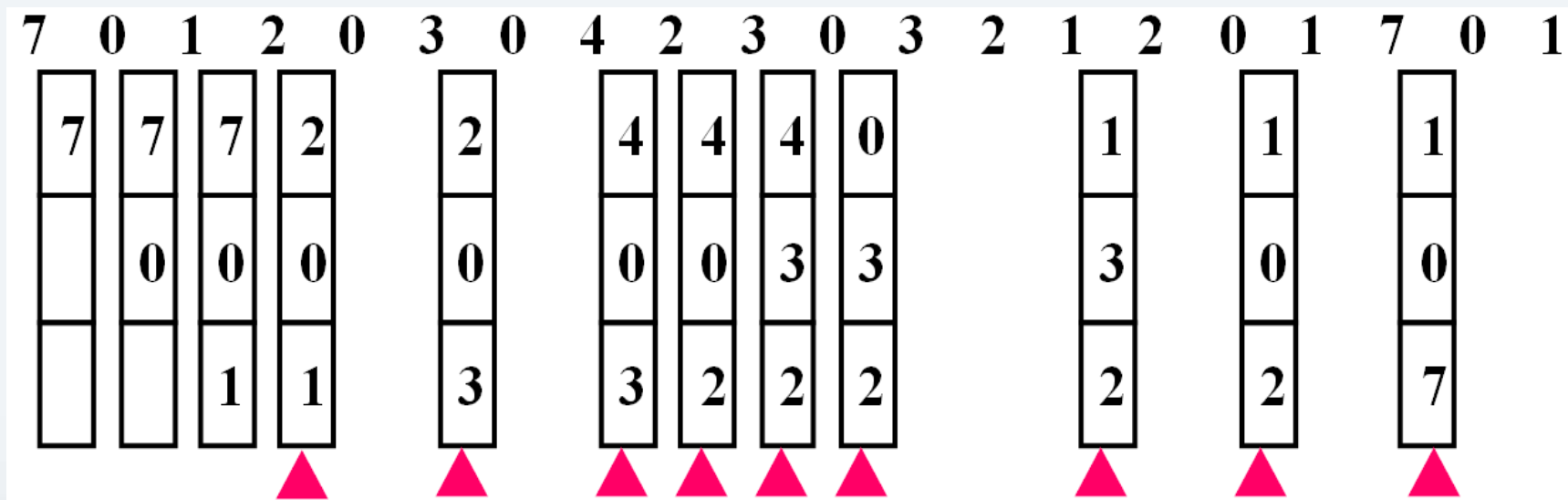
3.4.6 最近最少使用页面置换算法

- 根据程序局部性原理，那些刚被使用过的页面，可能马上还要被使用，而在较长时间里未被使用的页面，可能不会马上使用到。
- 由于需要记录页面使用时间的先后关系频繁度，硬件开销太大。
- 近似实现：选择内存中最近一段时间里较久未被访问的页面予以淘汰。性能接近最佳置换算法。

3.4.6 最近最少使用页面置换算法

例：（同前例）

- 利用LRU页面置换算法时的置换图：



共发生9次页面置换

算法实现的过程与形式

- LRU的实现困难：如何发现并记录最近最少使用的页面？
- 简单的硬件实现形式：每个页表项保存一个计数器，记录访问修改次数
- 高效的硬件实现形式：用 $N \times N$ （ N 为页帧数）矩阵记录页面使用情况
- 软件模拟硬件实现形式：NFU算法和老化算法

使用矩阵的LRU

任何时刻，二进制数值最小的行就是最近最少使用的

0 1 2 3 2 1 0 3 2 3

Page				
	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

Page				
	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

(b)

Page				
	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

(c)

Page				
	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

(d)

Page				
	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	1
3	1	1	0	0

(e)

0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

(f)

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

(g)

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

(h)

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

(i)

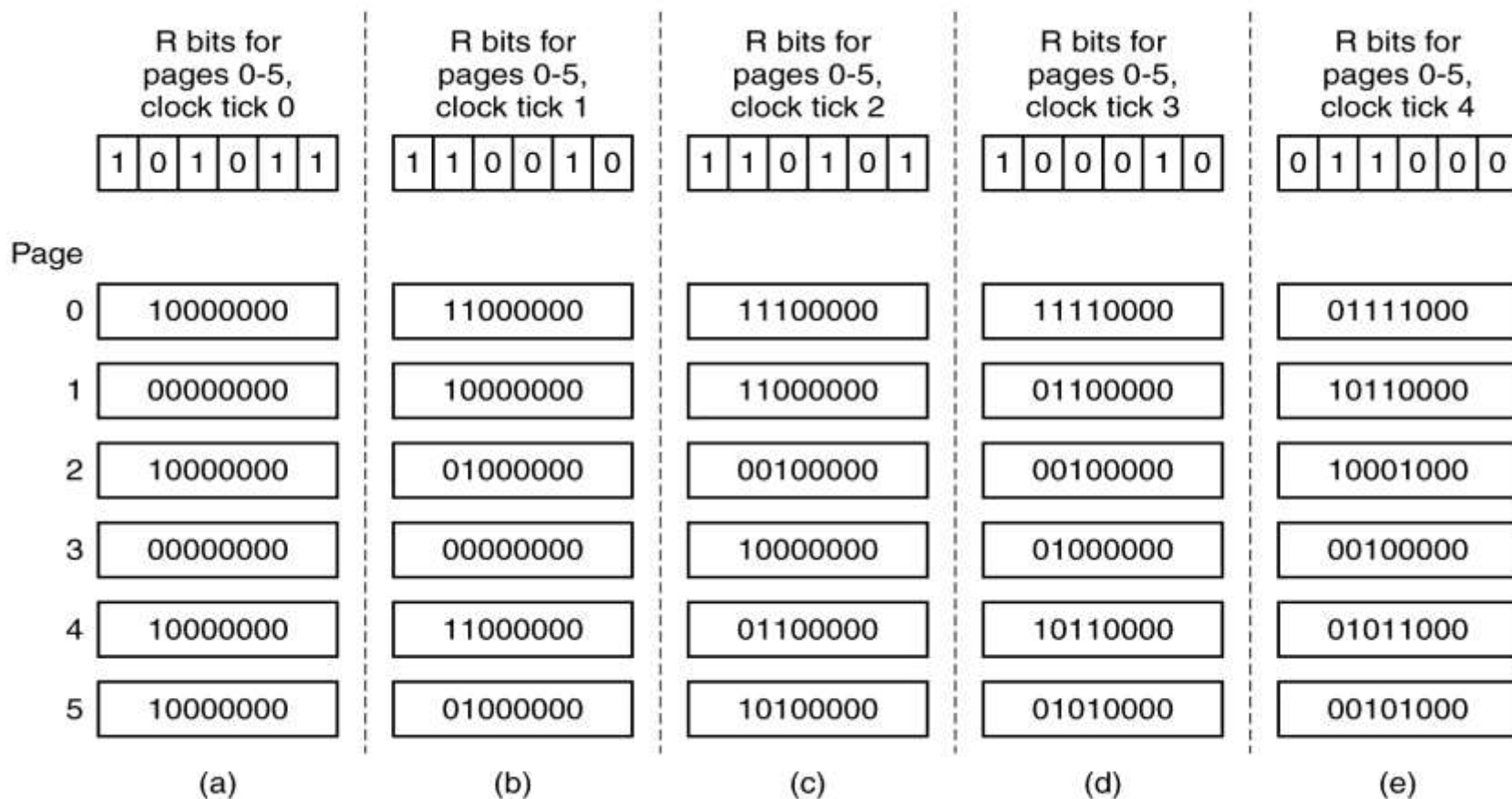
0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

(j)

3.4.7 用软件模拟LRU

- 每一页面与一个软件计数器相联，计数器初值为 0
- 时钟中断时，计数器右移1位，再将R加到计数器的最左端
- 发生页面失效时，将淘汰计数器最小的页面

软件模拟LRU的老化算法



3.4.7 用软件模拟LRU

不足：

- 每个时钟滴答中只记录了一位，无法区分一个滴答内在较早和较晚时间被访问的页面。
- 计数器位数有限。

3.4.8 工作集页面置换算法

- 抖动现象：经常将调出的页再调入
- 内存中引入过多的进程，造成多道程序度过高
- 每个进程并发执行时对内存进行访问，经常出现缺页情况
- 启动置换策略将某个（某些）页面换出，调入新页
- 继续访问时发现，又用到刚才调出的页面，又引起缺页中断

3.4.8 工作集页面置换算法

- 多道程序环境下，CPU的利用率开始时是随并发进程个数的增加而增加的，但进程个数超过一定数额时，CPU的利用率反而急剧下降——系统进入抖动。这就是为什么操作系统都必须限制进程总数的原因之一。操作系统应当采用较好的折衷，既要使CPU的利用率保持最佳，又不要使系统进入抖动。这是一个很难解决的问题。为此有必要借助于工作集模型。

3.4.8 工作集页面置换算法

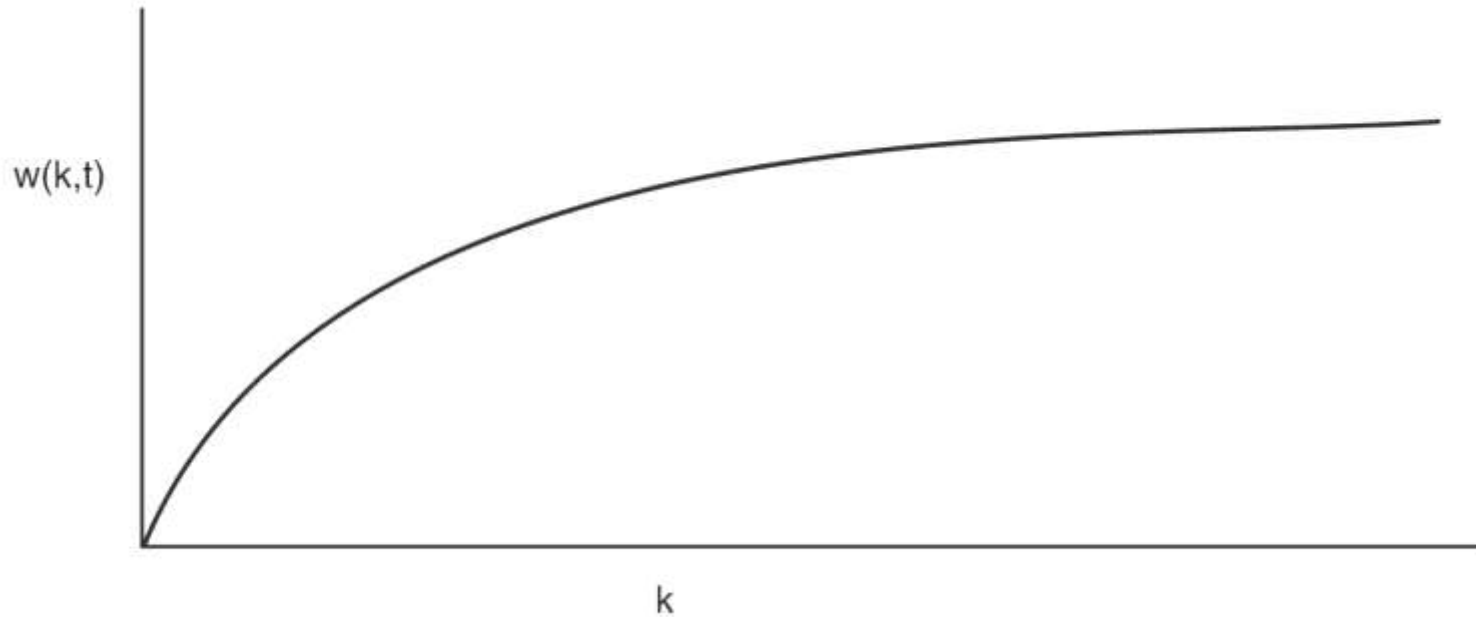
- 工作集理论是在1968年由Denning提出并推广的。Denning认为程序在运行时对页面的访问是不均匀的：即往往在某段时间内的访问仅局限于较少的页面。如果能够预知程序在某段时间间隔内要访问哪些页面，并能提前将它们调入内存，将会大大降低缺页率，减少置换工作，提高CPU的利用率。

3.4.8 工作集页面置换算法

- 何谓工作集——所谓工作集是指在某段时间间隔 T 内，进程实际要访问的页面集合。
- 核心思想：当页面失效发生时，淘汰一个不在工作集中的页面。
- 如何确定在某时刻，哪些页面不在工作集中？

工作集页面置换算法

- Denning认为，虽然程序只需少量的几页已在内存就可运行，但为使程序能有效地运行，较少地产生缺页，就必须使程序的工作集全部在内存中。然而，由于我们无法预知程序在不同时刻将访问哪些页面，因而只能像置换算法那样，**利用程序过去某段时间内的行为，作为程序在将来某段时间内行为的近似。**



函数 $w(k, t)$ 是在时刻 t 的工作集的大小。即工作集是指在进程运行中距时刻 t 最近的 k 次访问主存所涉及到的那些页面的集合。

3.4.8 工作集页面置换算法

例：

26157775162341234443434441327

| |t1 | |t2

$$w(10, t1) = \{1, 2, 5, 6, 7\}$$

$$w(10, t2) = \{3, 4\}$$

工作集模型简介

- 满足理论要求的工作集形式——最近K次内存访问所使用过的页面集合
- 满足实现要求的工作集形式——过去T秒实际时间中所使用过的页面集合
- 工作集算法的技术实现变形——基于时钟替换形式的工作集页面替换算法

工作集模型



- 扫描所有页面检查R位
- 若 ($R = 1$): 将上次使用时间为当前实际时间
- 若 ($R = 0$ 且 $\text{Age} > T$), 替换
- 若 ($R = 0$ 且 $\text{Age} \leq T$), 记住最小时间
- 最坏情况: 随机替换

3.4.9 工作集时钟页面置换算法

- 当缺页中断发生后，有时需要扫描整个页表才能确定被淘汰的页面，因此基本工作集算法是比较费时的。
- WSClock（工作集时钟）算法：基于时钟算法，并且使用了工作集信息，实现简单，性能较好，所以在实际工作中得到了广泛应用。

3.4.9 工作集时钟页面置换算法

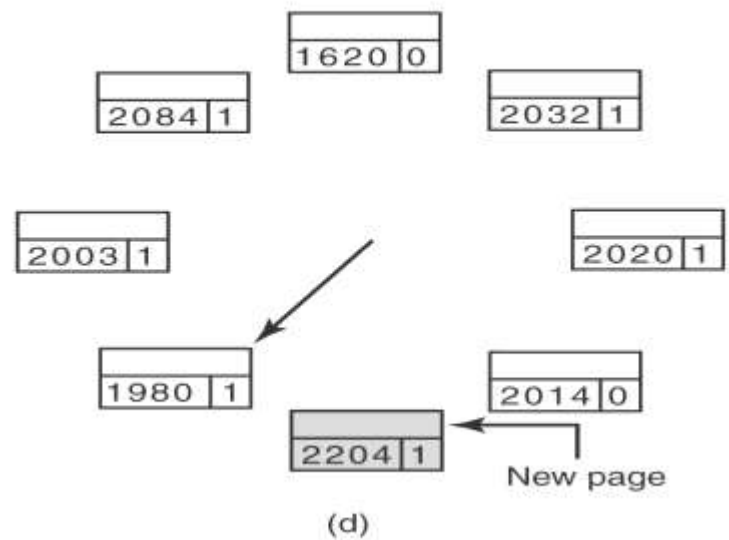
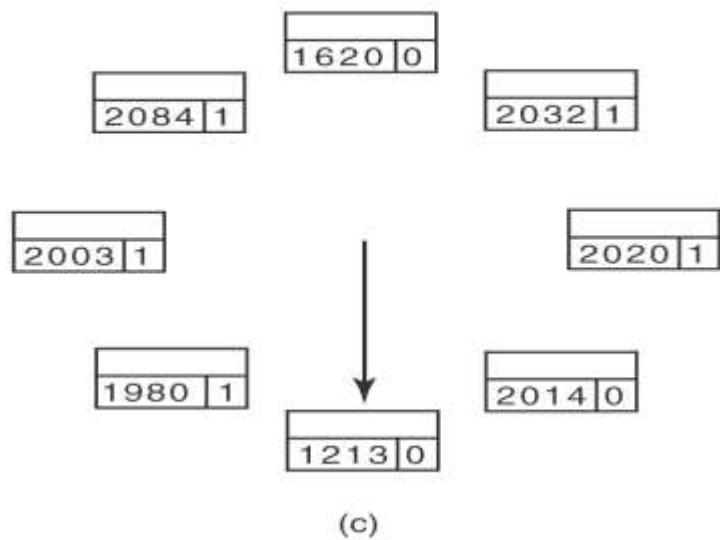
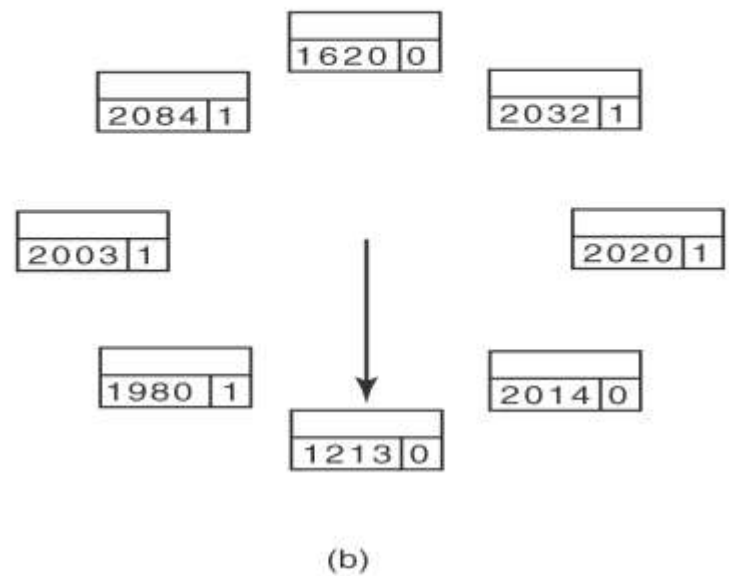
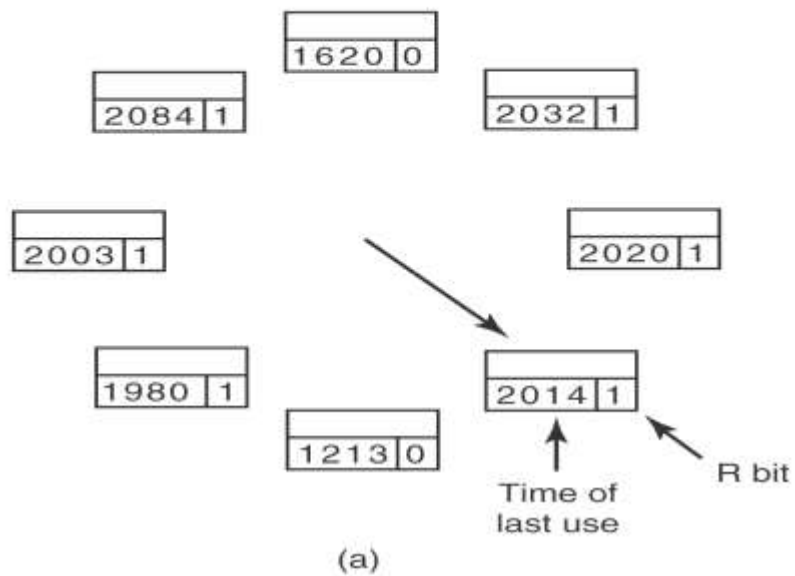
- 所需数据结构：以页帧为元素的循环表，包含上次使用时间域，R位，M位。
- 页面失效时，首先检查指针指向的页面，若R位为1，则不适合淘汰。同时，将该页的R位置为0，指针指向下一个页面。

3.4.9 工作集时钟页面置换算法

若指针指向页面的R位为0:

- $\text{Age} > T$ 且 $M=0$, 不在工作集中, 淘汰该页, 调入新页。
- $\text{Age} > T$ 且 $M=1$, 写磁盘, 指针前进。
- $\text{Age} < T$: 在工作集中, 指针前进。

2204 Current virtual time



3.4.10 页面置换算法小结

	算法原理	算法性能	使用原则
最优算法	替换最久不会被使用的页面	最佳算法	无法实现
NRU算法	简单利用R、M位进行页面分类	粗糙设计	不实用
FIFO式算法	根据进入内存时间选择替换页面	常识推断	逐步形成现实科研的方法
LRU算法	局部性原理推断	接近最优的设计	高消耗、高性能
NFU&老化算法	软件模拟LRU	最可能的LRU	有效实现的现实算法
工作集算法/ 时钟算法	基于全局分析的局部性选择	最可信（看起来）的设计	最优秀的有效算法

【思考题】

- 某程序在内存中分配3块内存，初始为空，访问页的走向为2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2, 用OPT, LRU, FIFO和CLOCK算法分别计算缺页次数和缺页率。

缺页率的计算：

缺页率=缺页次数/访问串的访问次数

算法对比

	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
	↑	↑		↑	↑↑		↑↑			↑↑		
LRU	2	2	3	3	2	1	5	2	4	5	3	3
		3	2	2	1	5	2	4	5	3	2	5
				1	5	2	4	5	3	2	5	2
	↑	↑		↑	↑↑		↑↑		↑↑	↑↑		
FIFO	2	2	2	2	3	1	5	5	2	2	4	3
		3	3	3	1	5	2	2	4	4	3	5
				1	5	2	4	4	3	3	5	2
	↑	↑		↑	↑↑	↑↑	↑↑		↑↑		↑↑	↑↑

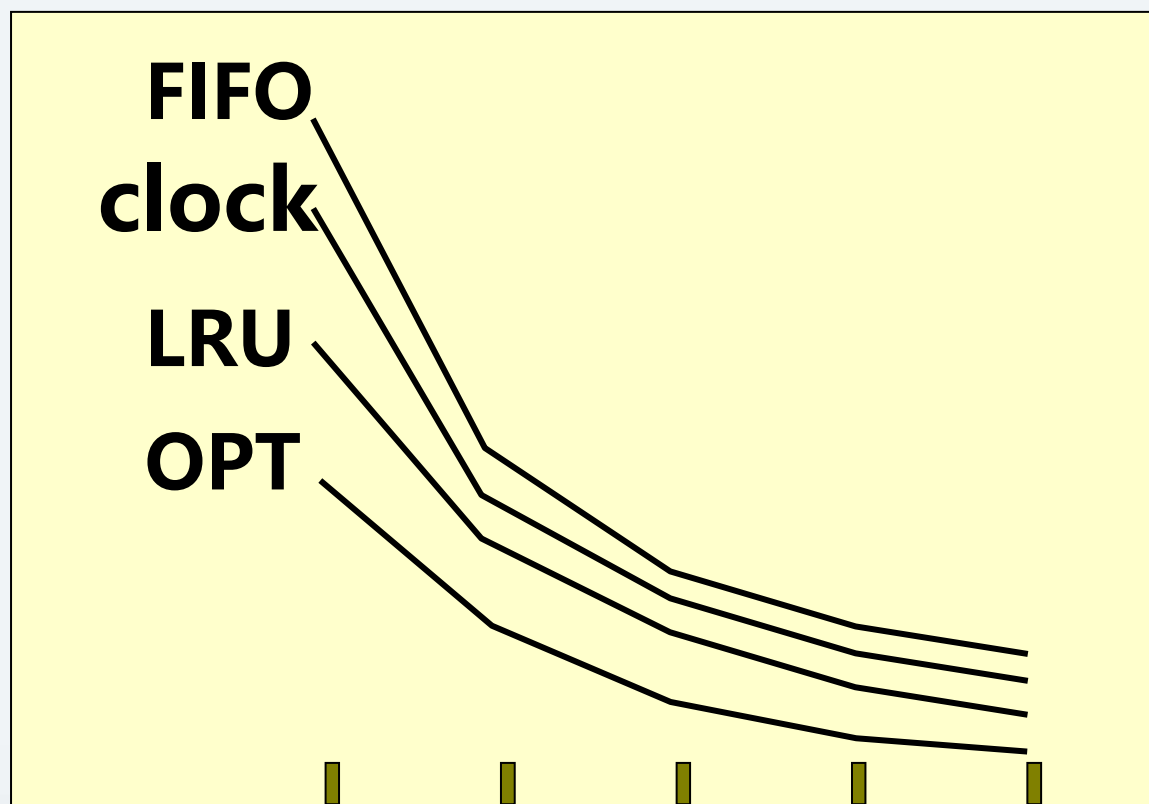
算法对比

	2	3	2	1	5	2	4	5	3	2	5	2
clock	2	2	2	2	5	5	5	5	5	5	5	5
		3	3	3	3	2	2	2	3	3	3	3
				1	1	1	4	4	4	2	2	2
	↑	↑		↑	↑↑	↑↑	↑↑		↑↑	↑↑		

在此，假设每个时钟周期只访问一个页面。

算法性能比较

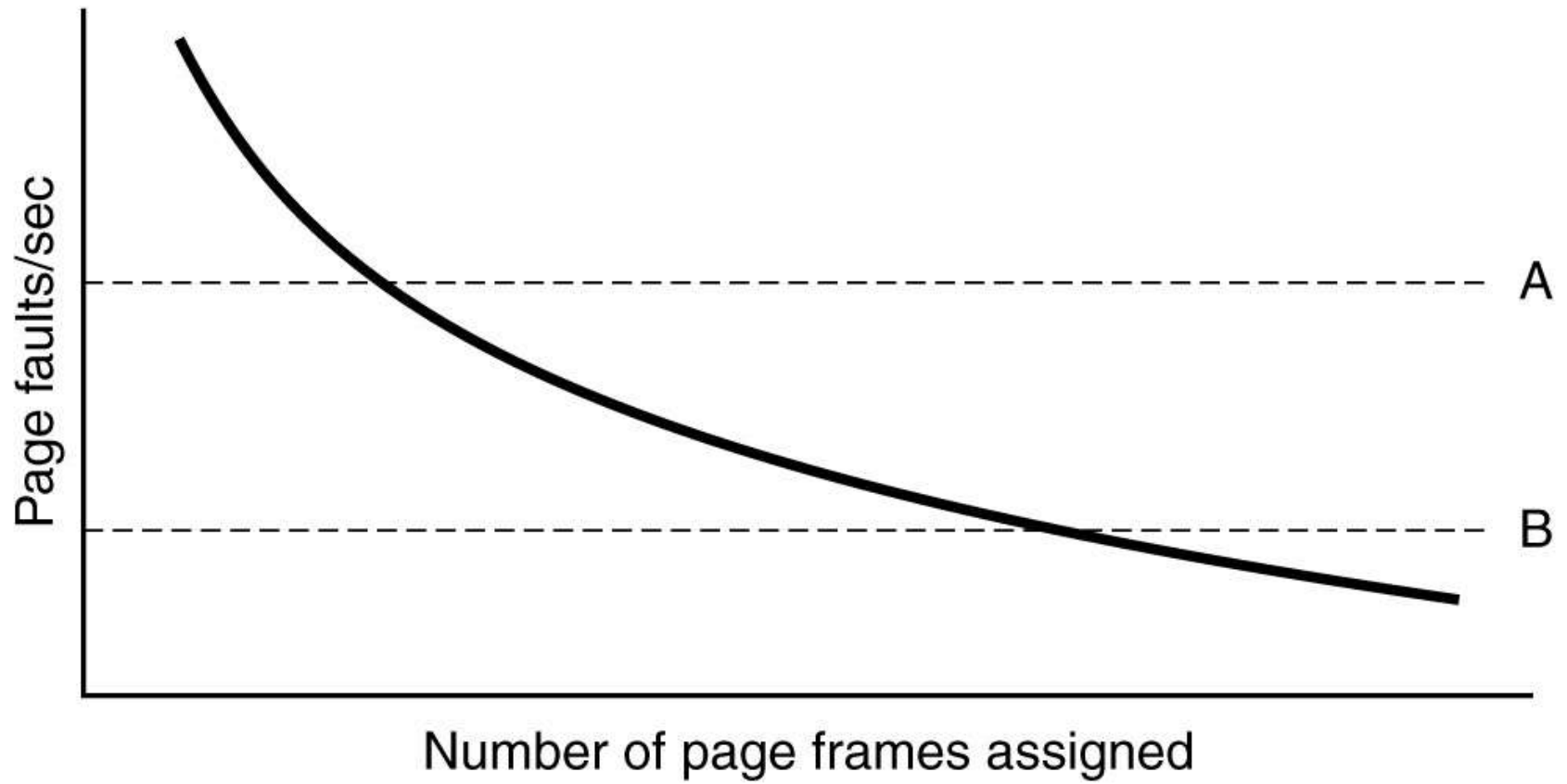
缺页率



6 8 10 12 14 分配的页面数

3.5 建立页面置换算法模型

- 缺页率 = “缺页次数 / 内存访问次数”
- 缺页率与分配给进程的内存页帧数目的关系：
 - 数目越多，缺页率越低
 - 页帧数目的下限应该是一条指令及其操作数可能涉及的页面
 - 数目的上限应该是足以保证每条指令都能被执行



Belady异常

- Belady现象：采用FIFO算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多，缺页率反而提高的异常现象。

Belady 异常

- 描述：一个进程P要访问M个页，OS分配N个内存页面给进程P；对一个访问序列S，发生缺页次数为 $PE(S, N)$ 。当N增大时， $PE(S, N)$ 时而增大，时而减小。
- 原因：FIFO算法的置换特征与进程访问内存的动态特征是矛盾的，即被置换的页面并不是进程不会访问的。

Belady 现象的例子:

进程P有5页程序。访问页的顺序为: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5。

(1) 如果在内存中分配3个页面, 则缺页情况如下: 12次访问中有缺页9次;

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	1	2	5	5	5	3	4	4
页 1		1	2	3	4	1	2	2	2	5	3	3
页 2			1	2	3	4	1	1	1	2	5	5
缺页	x	x	x	x	x	x	x	√	√	x	x	√

Belady 现象的例子:

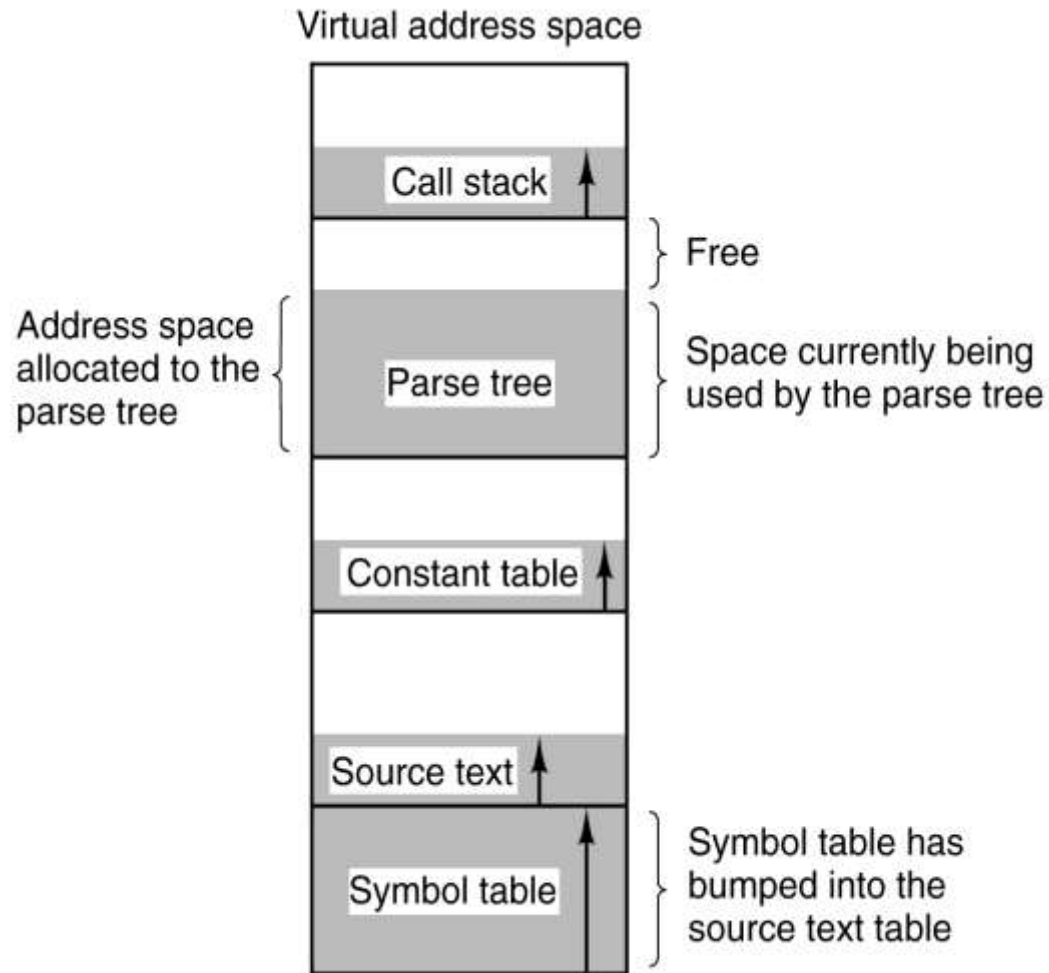
(2) 如果在内存中分配4个页面, 则缺页情况如下:
12次访问中有缺页10次;

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	4	4	5	1	2	3	4	5
页 1		1	2	3	3	3	4	5	1	2	3	4
页 2			1	2	2	2	3	4	5	1	2	3
页 3				1	1	1	2	3	4	5	1	2
缺页	x	x	x	x	√	√	x	x	x	x	x	x

3.7 分段

- 分页：提高主存空间利用率。
- 分段：满足用户(程序员)编程和使用上的要求。

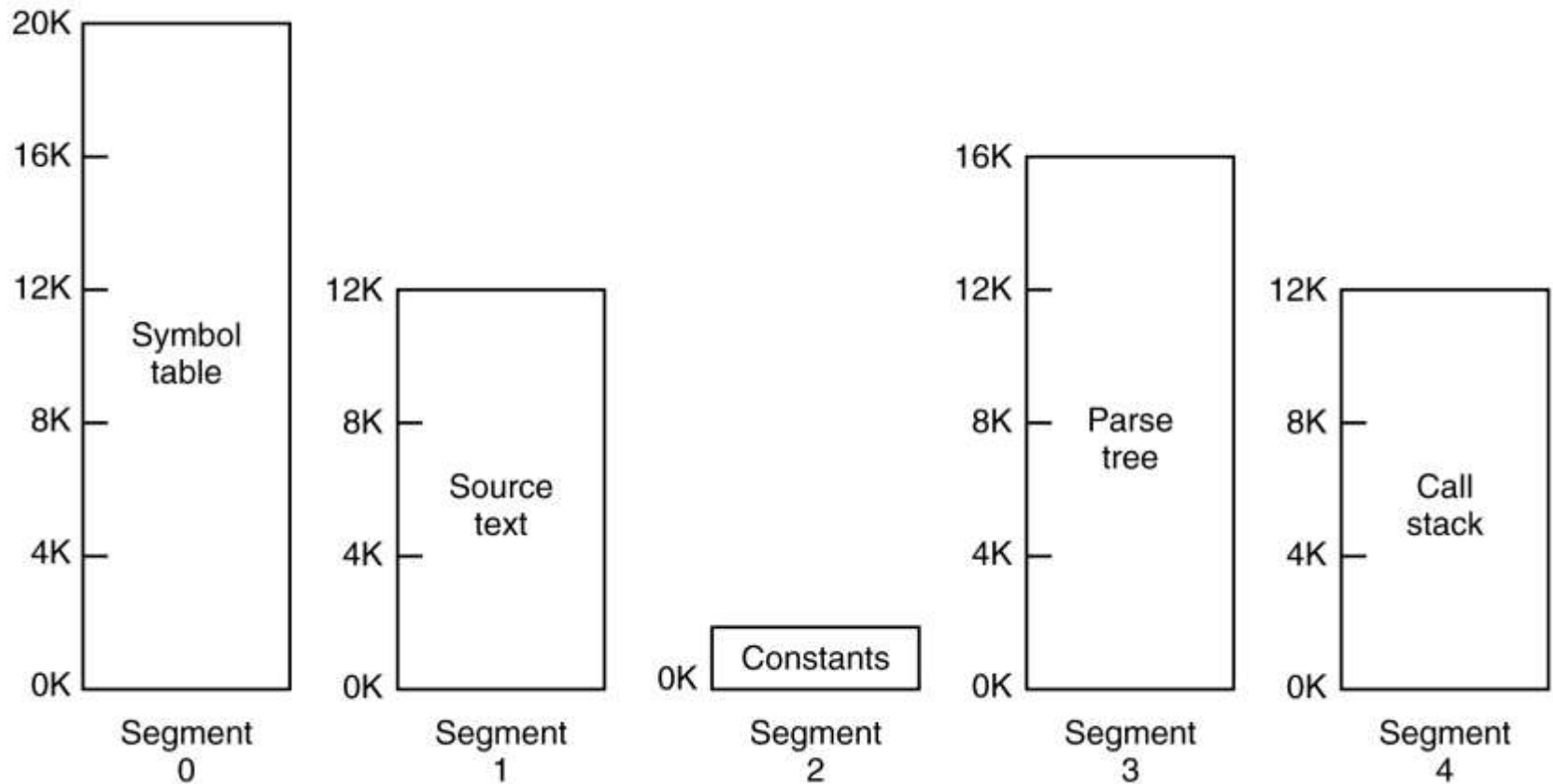
3.7 分段



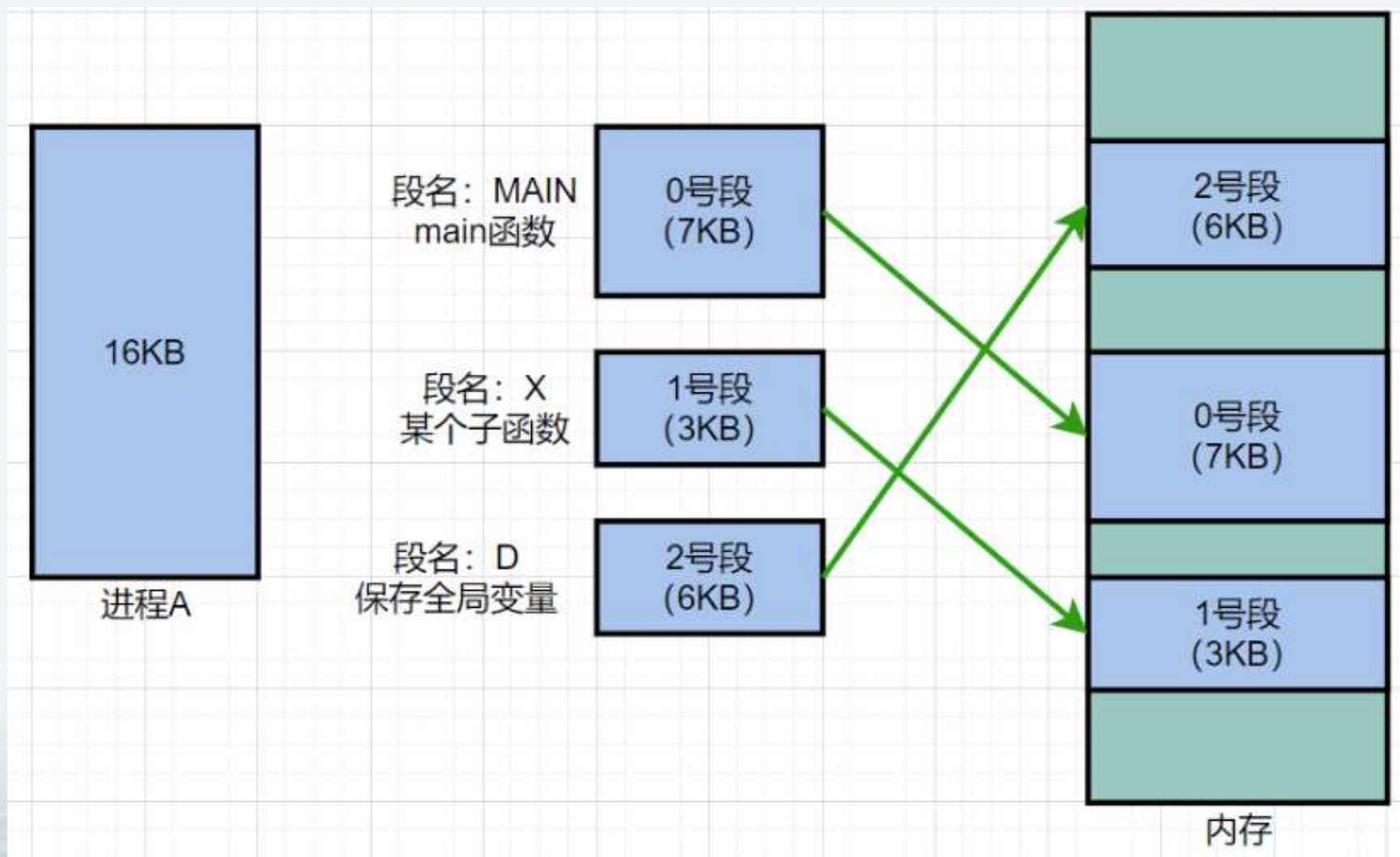
3.7 分段

- 将程序的地址空间按内容或过程（函数）关系划分为若干个段，每段有自己的名字。以段为单位分配内存，然后通过地址重定位机构把段式虚拟地址转换为实际的内存物理地址。

3.7 分段



分段

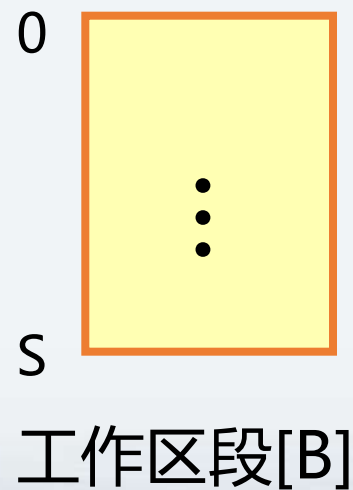
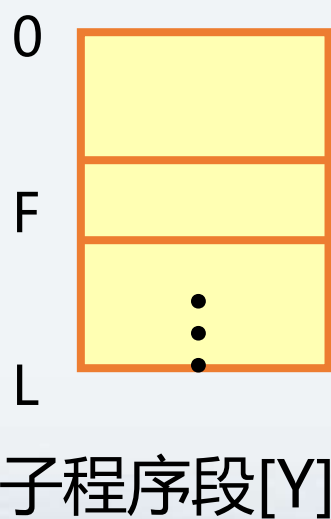
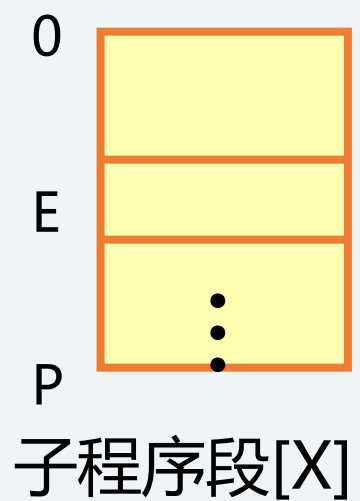
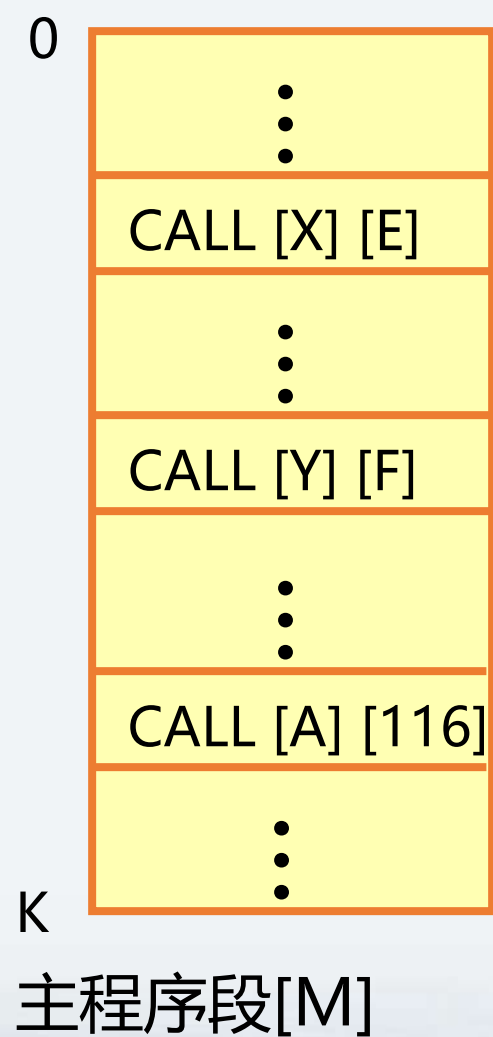


3.7 分段

- 段由一个从0到最大的线性地址序列构成。
- 不同的段长度可以不同。
- 段的长度在运行期间可以变化。
- 每个段都构成一个独立的地址空间，它们可以独立地增长或减小而不会影响其他的段。

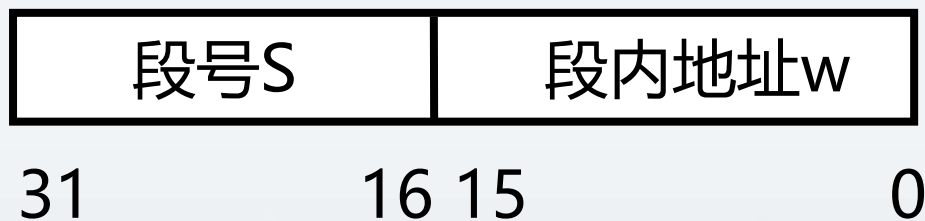
分段的好处

- 段可以不相等，动态尺寸变化
- 简化不断增长的数据结构
- 允许程序独立地改变或重新编译
- 有助于进程间的共享
- 有助于进程间的保护



分段系统的基本原理

- 每个段定义了一组逻辑信息。如：主程序段、子程序段、数据段等。
- 两维逻辑地址：段号 + 段内地址。
- 地址结构：段号S + 段内地址W



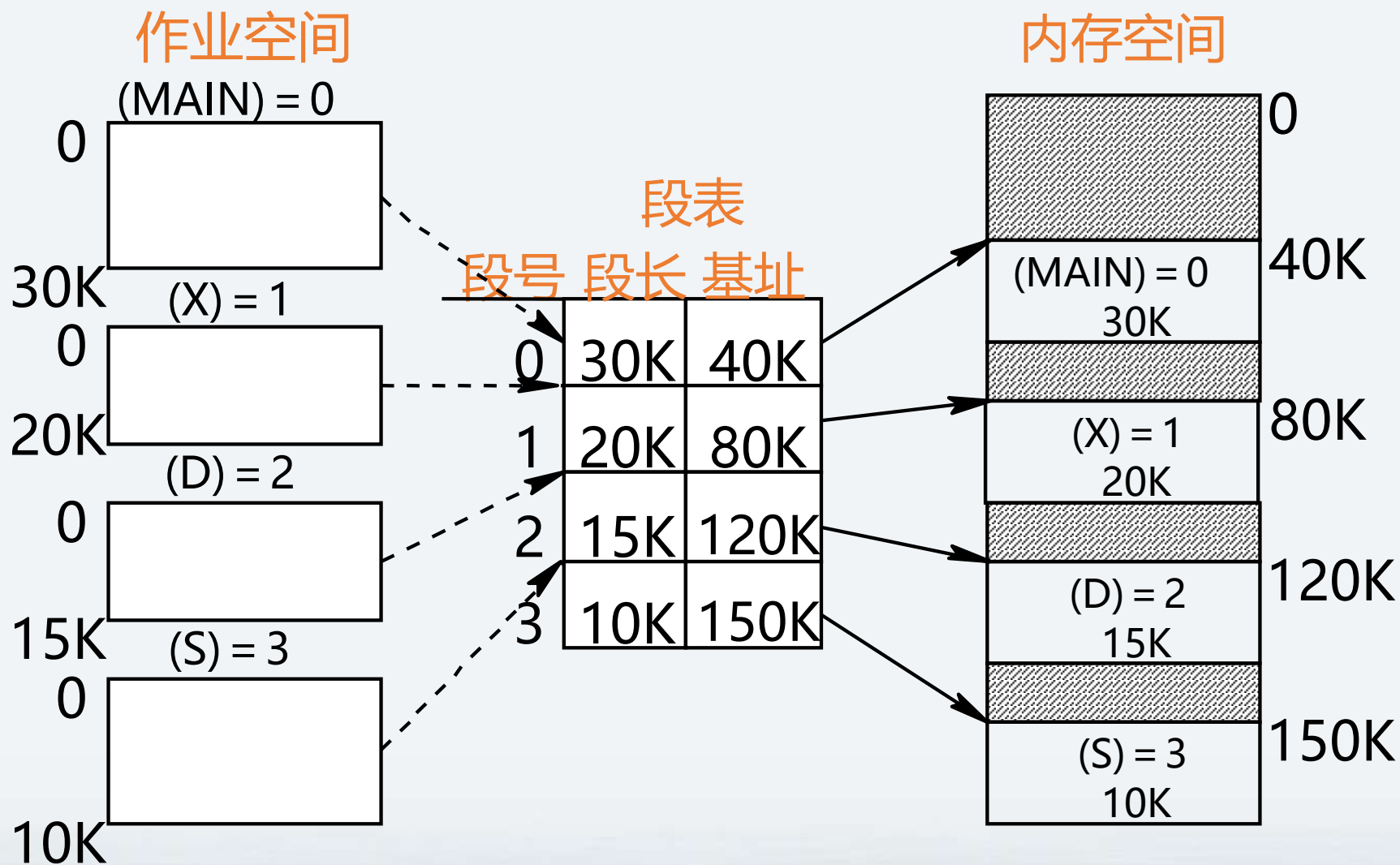
(一个作业最多 $2^{16}=64K$ 个段，每段最大长度 $2^{16}=64KB$)

段表

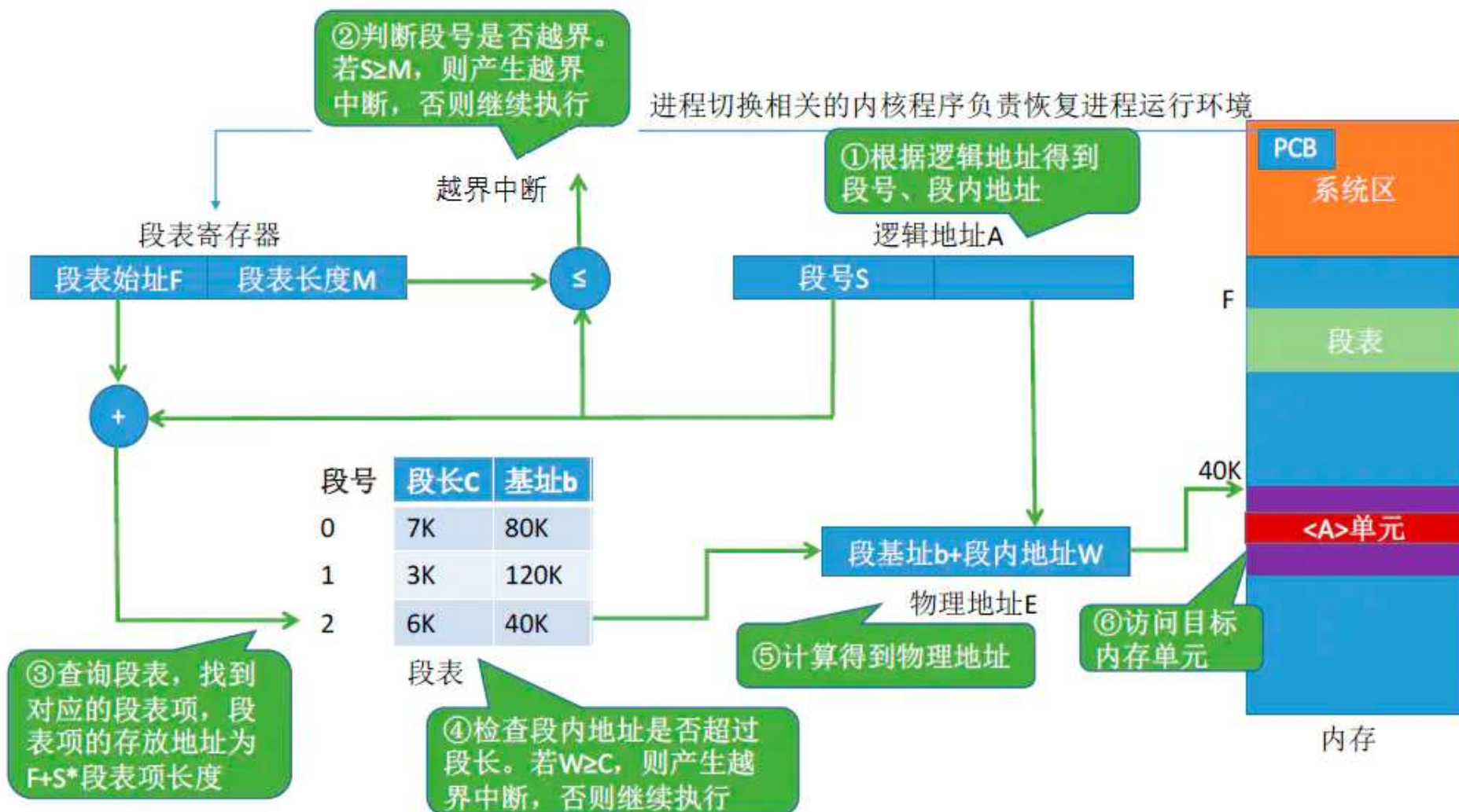
- 每个进程一张段表，用以实现地址变换，程序的每一个段在段表中占用一个表目。
- 包括：段的长度、段的首址(基址)信息等。

段号	段长度	段首址
0	20K	58K
1	110K	100K
2	140K	260K

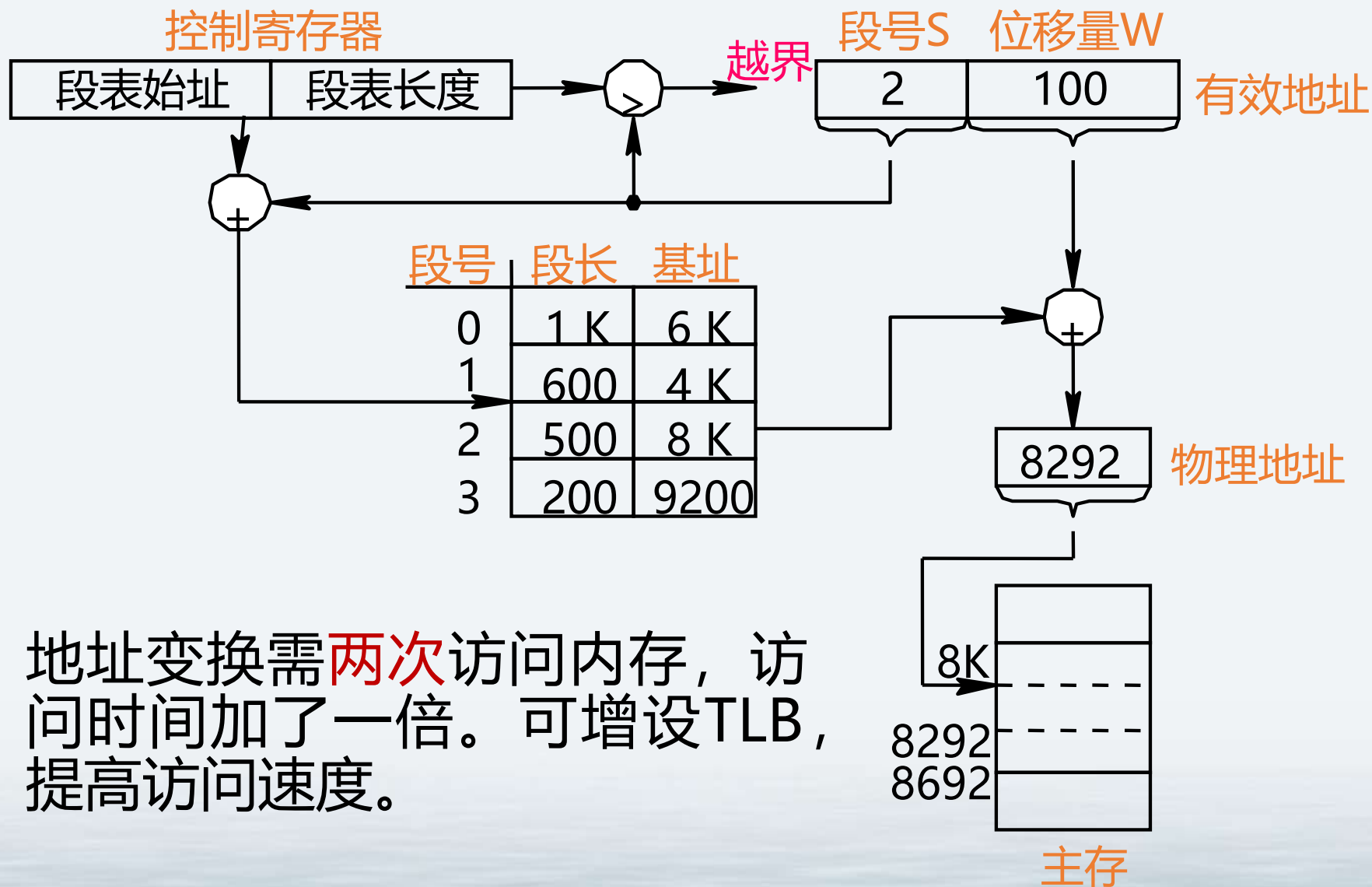
利用段表实现地址映射



地址变换

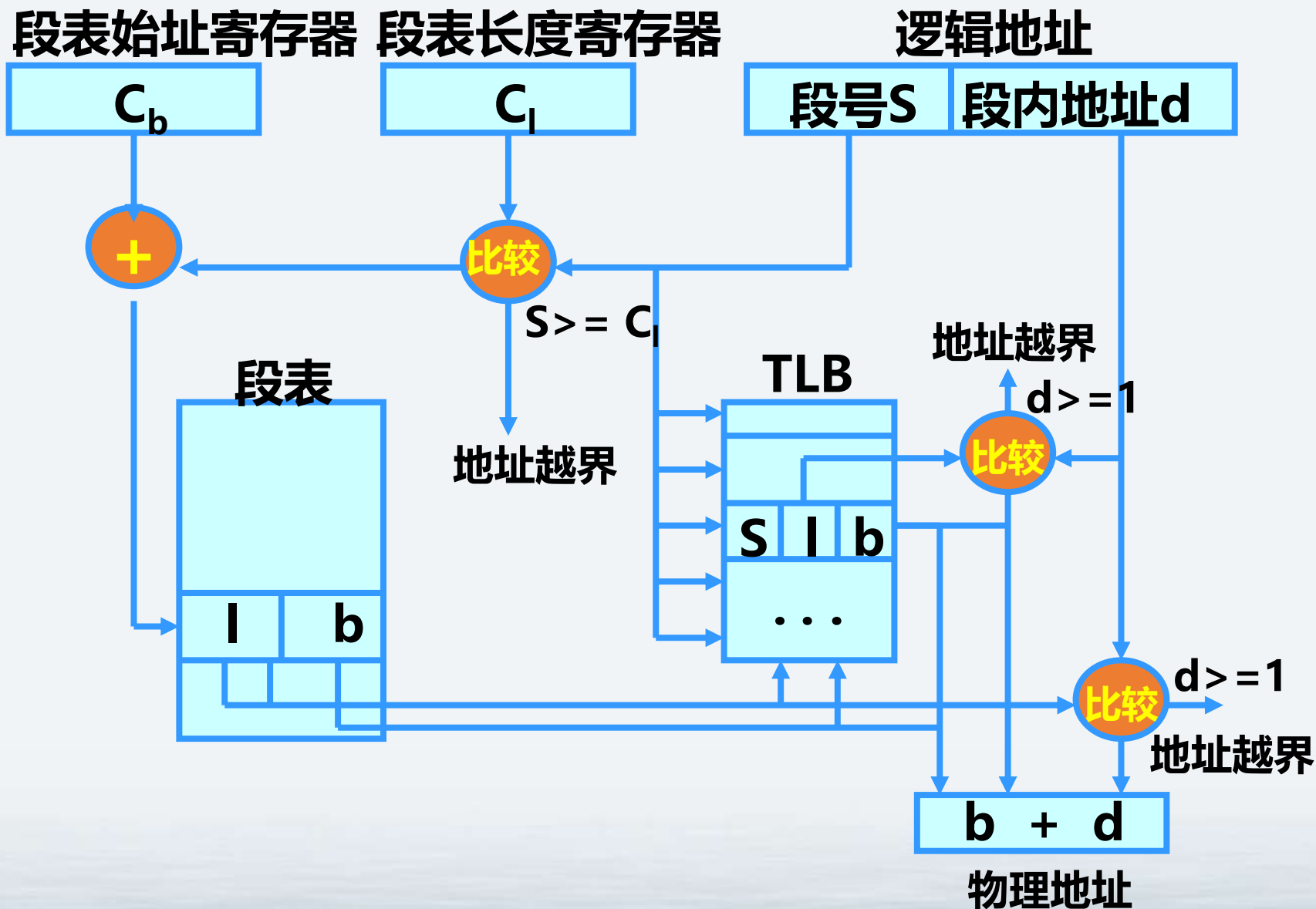


地址变换机构



地址变换需**两次**访问内存，访问时间加了一倍。可增设TLB，提高访问速度。

地址映射及存储保护机制



练习

例：有一段表如右图所示，试分别求逻辑地址（2, 88）和逻辑地址（4, 100）所对应的物理地址。

段号	基地址	段长
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

分段与分页技术的比较

- 相似：二者在内存中都不是整体连续的，都要通过地址映射机构将逻辑地址映射到物理内存中。
- 不同：
 - “页”是信息的“物理”单位，大小固定。“段”是信息的逻辑单位，即它是一组有意义的信息，其长度不定。
 - 页的大小是由系统固定的，在一个系统中所有页的大小是一样的，只能有一种大小；段的长度因段而异，取决于用户所编写的程序。
 - 分页的作业的地址空间是单一的线性地址空间，分段作业的地址空间是二维的。

3.7 分段

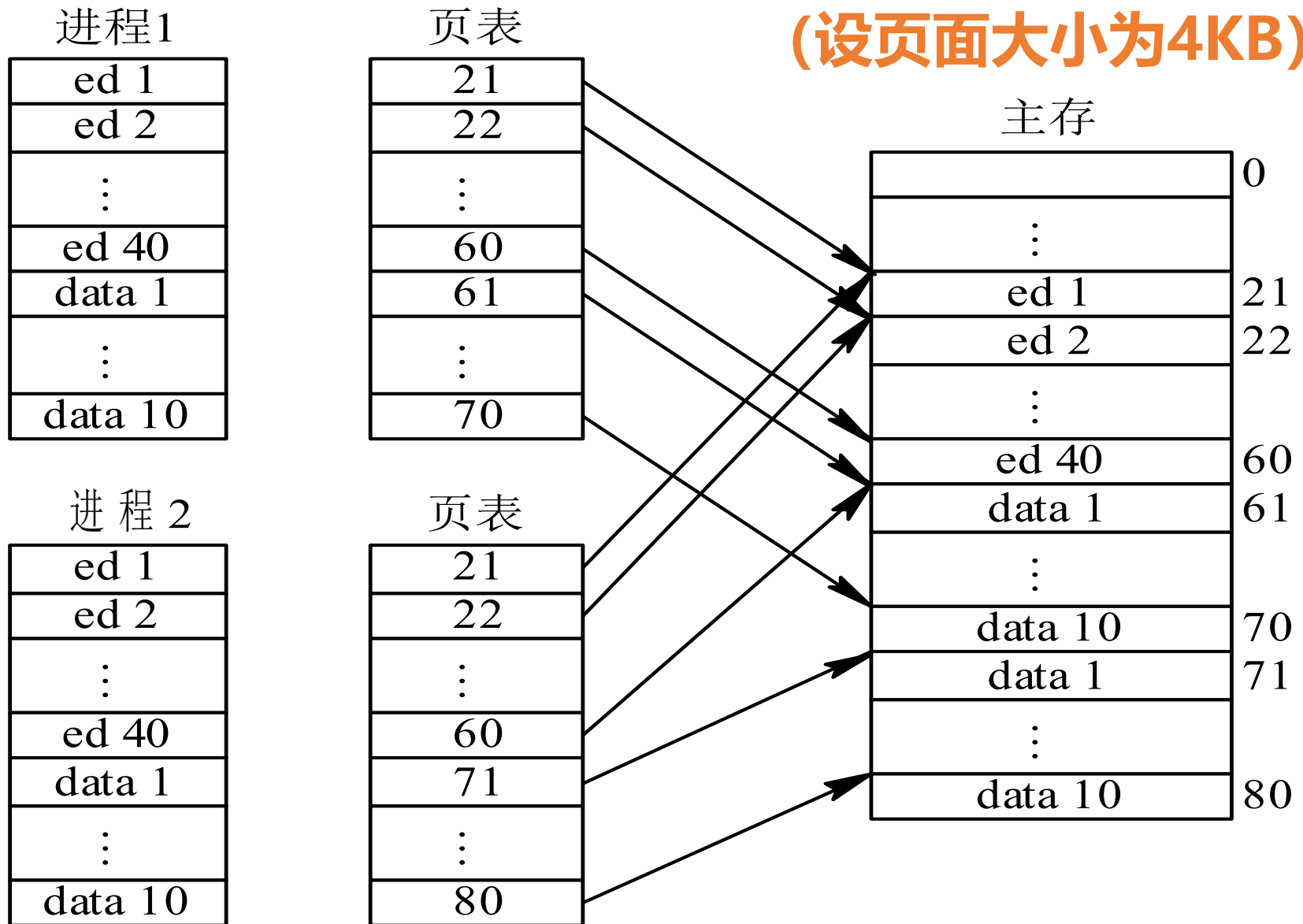
- 分段系统的一个突出优点是易于实现段的共享，即允许若干个进程共享一个或多个分段，且对共享段的保护也简单易行。分页系统虽然也能实现程序和数据的共享，但远不如分段系统方便。

3.7 分段

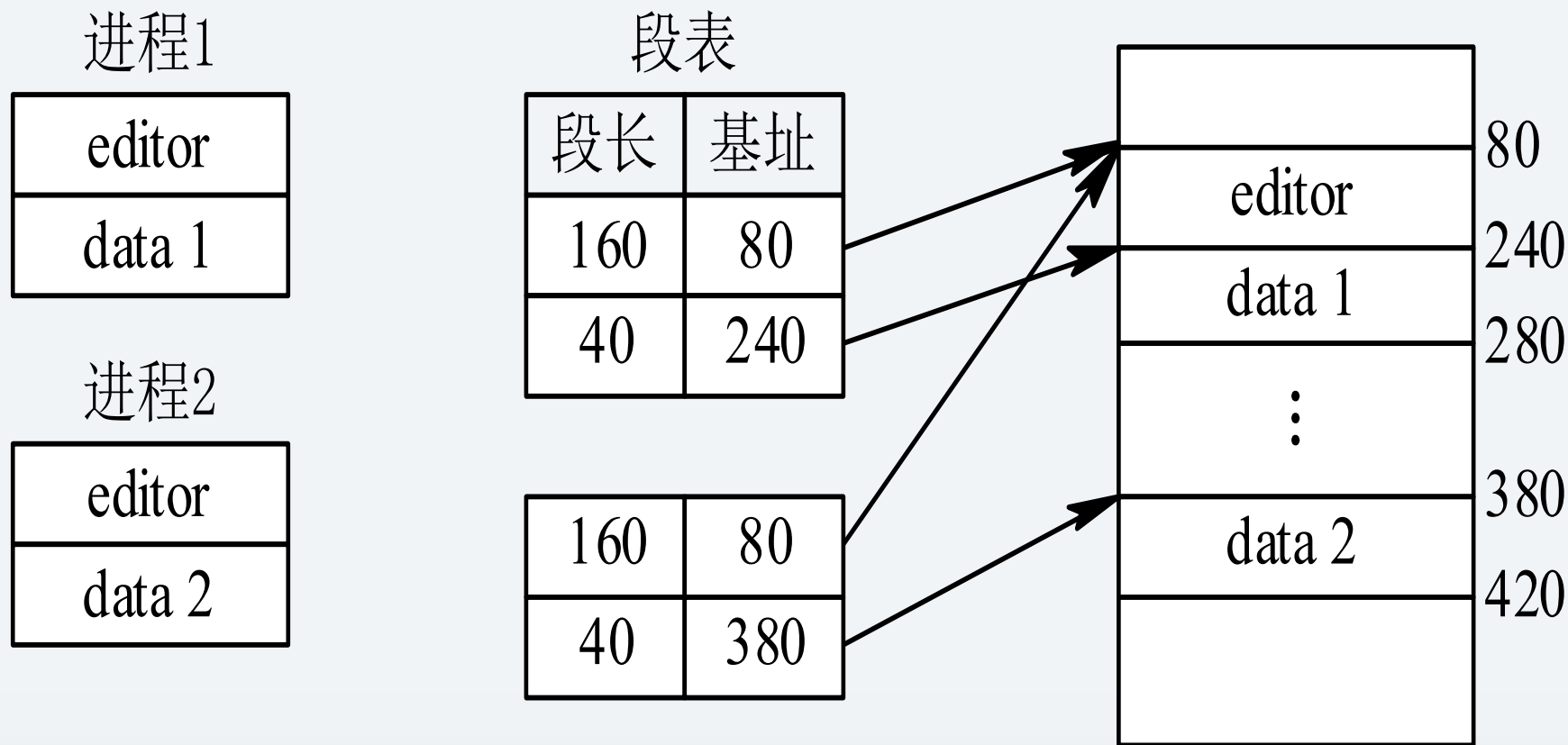
- 例：40个用户共享一文本编辑程序。该程序有160KB的代码和40KB的数据区。其中，160KB的代码为可重入代码（即允许多个进程同时访问的代码，但不允许任何进程对它进行修改）。

分页系统中共享 editor 的示意图

(设页面大小为4KB)



分段系统中共享editor的示意图



段的保护

越界中断处理

进程在执行过程中，有时需要扩大分段，如数据段。由于要访问的地址超出原有的段长，所以发越界中断。操作系统处理中断时，首先判断该段的“扩充位”，如可扩充，则增加段的长度；否则按出错处理。

段的保护

缺段中断处理

检查内存中是否有足够的空闲空间

①若有，则装入该段，修改有关数据结构，中断返回；

②若没有，检查内存中空闲区的总和是否满足要求，是则应采用紧缩技术，转 ①；否则，淘汰一（些）段，转①。

段页式存储管理方式

- 分页系统中，内零头得到了有效的抑制，外零头则完全被消除；因此，使用分页技术可以提高物理内存的利用率
- 分段系统中，动态数据结构、程序和数据共享、程序和数据保护等问题得到了妥善解决；因此，分段技术有利于模块化程序设计
- 段页技术汲取了分页技术和分段技术的上述优点

段页式存储管理方式

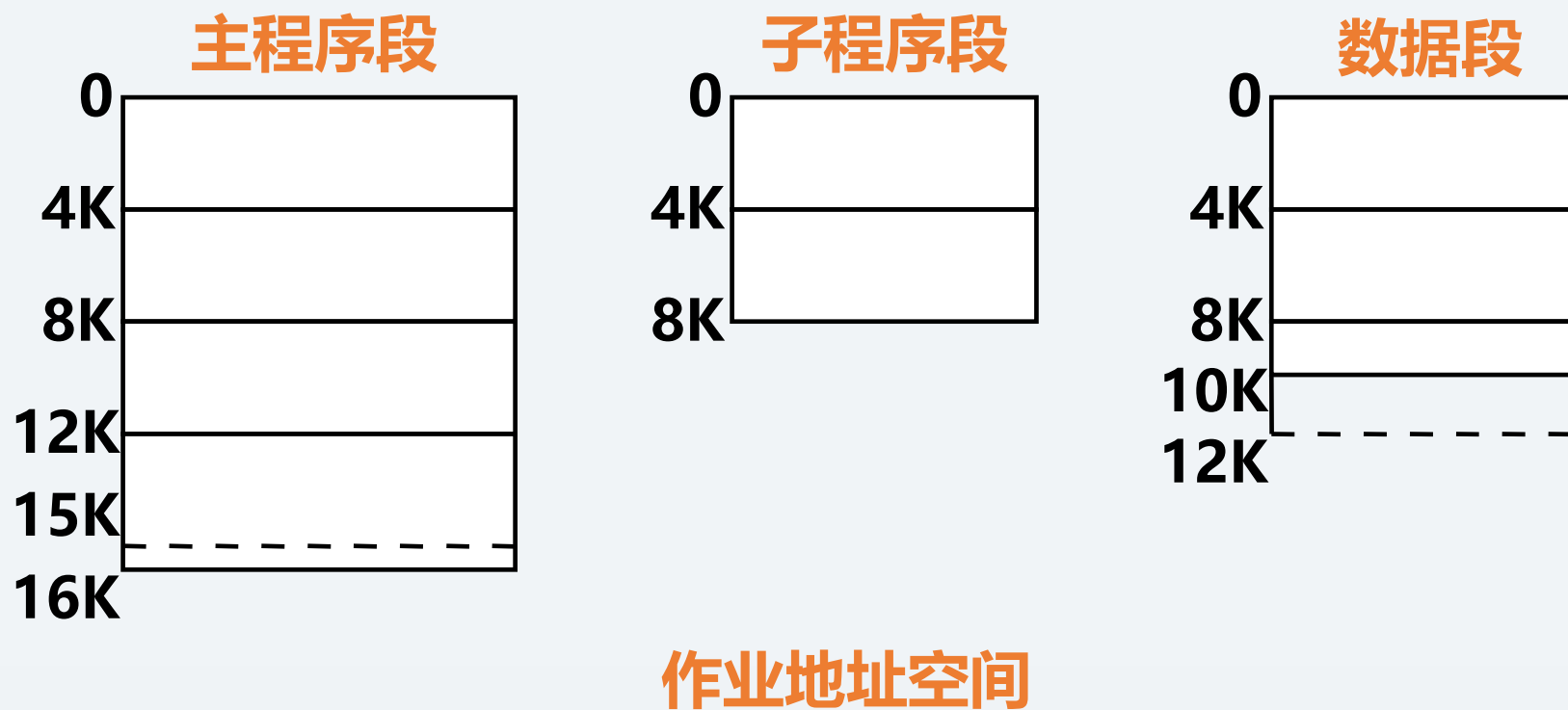
- 每个进程均被编程人员分割成多个Segment, 每个段又被系统分割成多个Page;
- 相应地, 物理内存被系统划分成多个Frame;
- 当一个进程被装入物理内存时, 系统为该进程的每个段的各页面独立地分配一个Frame;
- 一个进程的同一段的多个页面不必存放在连续的多个Frame中

段页式存储管理方式

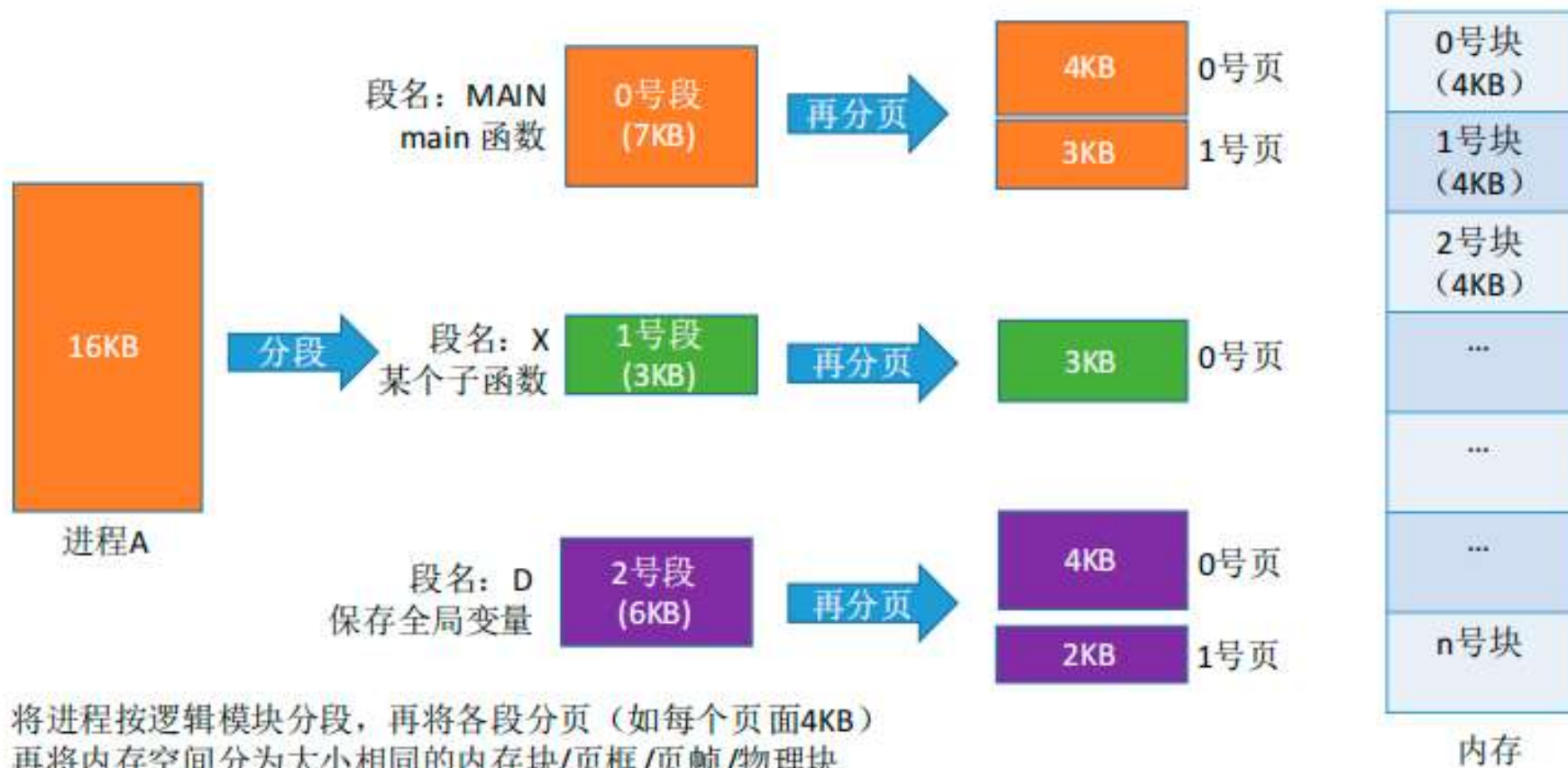
地址结构：分三部分

段号(S)	段内页号(P)	页内地址(W)
-------	---------	---------

段页式存储管理方式



段页式存储管理方式

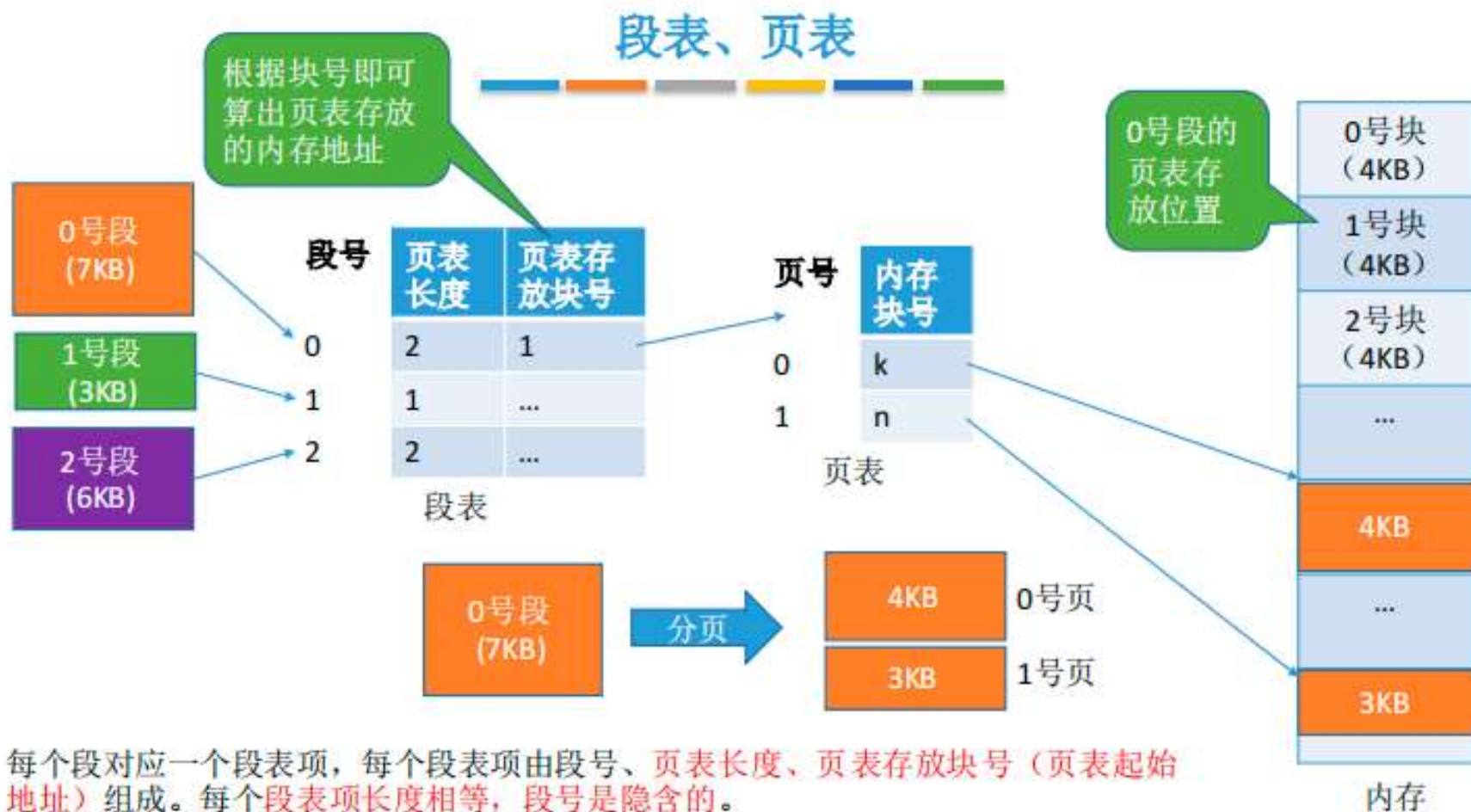


将进程按逻辑模块分段，再将各段分页（如每个页面4KB）
再将内存空间分为大小相同的内存块/页框/页帧/物理块
进程前将各页面分别装入各内存块中

段页式存储管理方式

- 段表：记录了每一段的页表始址和页表长度。
- 页表：记录了逻辑页号与内存块号的对应关系（每一段有一个，一个程序可能有多 个页表）。
- 内存分配管理：同页式管理。

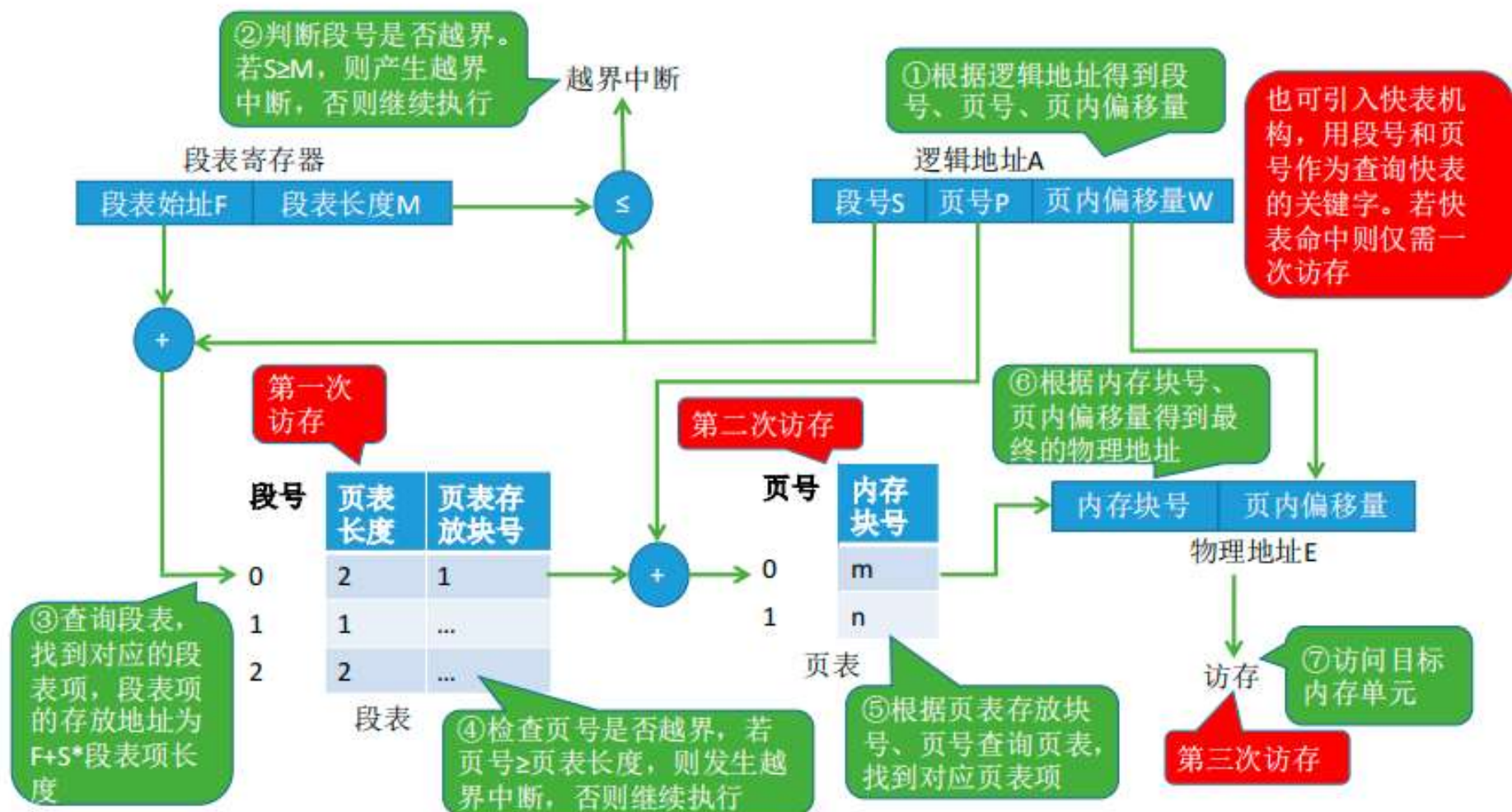
地址映射



每个段对应一个段表项，每个段表项由段号、页表长度、页表存放块号（页表起始地址）组成。每个段表项长度相等，段号是隐含的。

每个页面对应一个页表项，每个页表项由页号、页面存放的内存块号组成。每个页表项长度相等，页号是隐含的。

地址变换过程



段页式存储管理方式

- 地址变换需三次访问内存：一次访问段表、一次访问页表、一次访问指令或数据，访问时间加了两倍。可增设TLB，提高访问速度。

小结

段页式管理

分段+分页

将地址空间按照程序自身的逻辑关系划分为若干个段，再将各段分为大小相等的页面

将内存空间分为与页面大小相等的一个个内存块，系统以块为单位为进程分配内存

逻辑地址结构：（段号，页号，页内偏移量）

段表、页表

每个段对应一个段表项。各段表项长度相同，由段号（隐含）、**页表长度、页表存放地址** 组成

每个页对应一个页表项。各页表项长度相同，由页号（隐含）、页面存放的内存块号 组成

地址变换

1. 由逻辑地址得到段号、页号、页内偏移量
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. **根据段表中记录的页表长度，检查页号是否越界**
5. 由段表中的页表地址、页号得到查询页表，找到相应页表项
6. 由页面存放的内存块号、页内偏移量得到最终的物理地址
7. 访问目标单元

访问一个逻辑地址所需访存次数

第一次——查段表、第二次——查页表、第三次——访问目标单元

可引入快表机构，以段号和页号为关键字查询快表，即可直接找到最终的目标页面存放位置。引入快表后仅需一次访存

【思考题】

下图是一种段页式管理配置方案。

- (1) 根据给出的虚地址写出实地址。
- (2) 描述地址映像过程。
- (3) 从虚地址要经过几次访问主存？
- (4) 一般采用什么设施提高映像速度？

指令寄存器地址部分

段表首地址寄存器

6	4	337
---	---	-----

1000

页表

6000	17000
6001	13000
6002	17600
6003	16000
6004	14000
6005	12000
6006	15000

页表

2000	45000
2001	42000
2002	44000
2003	47000
2004	46000
2005	43000
2006	41000

页表

5000	32000
5001	33000
5002	36000
5003	31000
5004	37000
5005	35000
5006	34000

段表

1000	3000
1001	4000
1002	7000
1003	2000
1004	5000
1005	56000
1006	6000
1007	7600

【问题的解】

(1) 主存实地址为14337

(2) 地址映像过程为：段号6映像到段表为1006，在段表1006表项查得页表地址为6000。进而查得页表项6004内容为14000（假设此即为物理页的首地址），加上位移量337即得实地址为14337。

(3) 共三次访问主存：第一次访问段表，第二次访问页表，第三次访问实地址。

(4) TLB，存放常用的页表项。