计算机基础-操作系统

内存管理

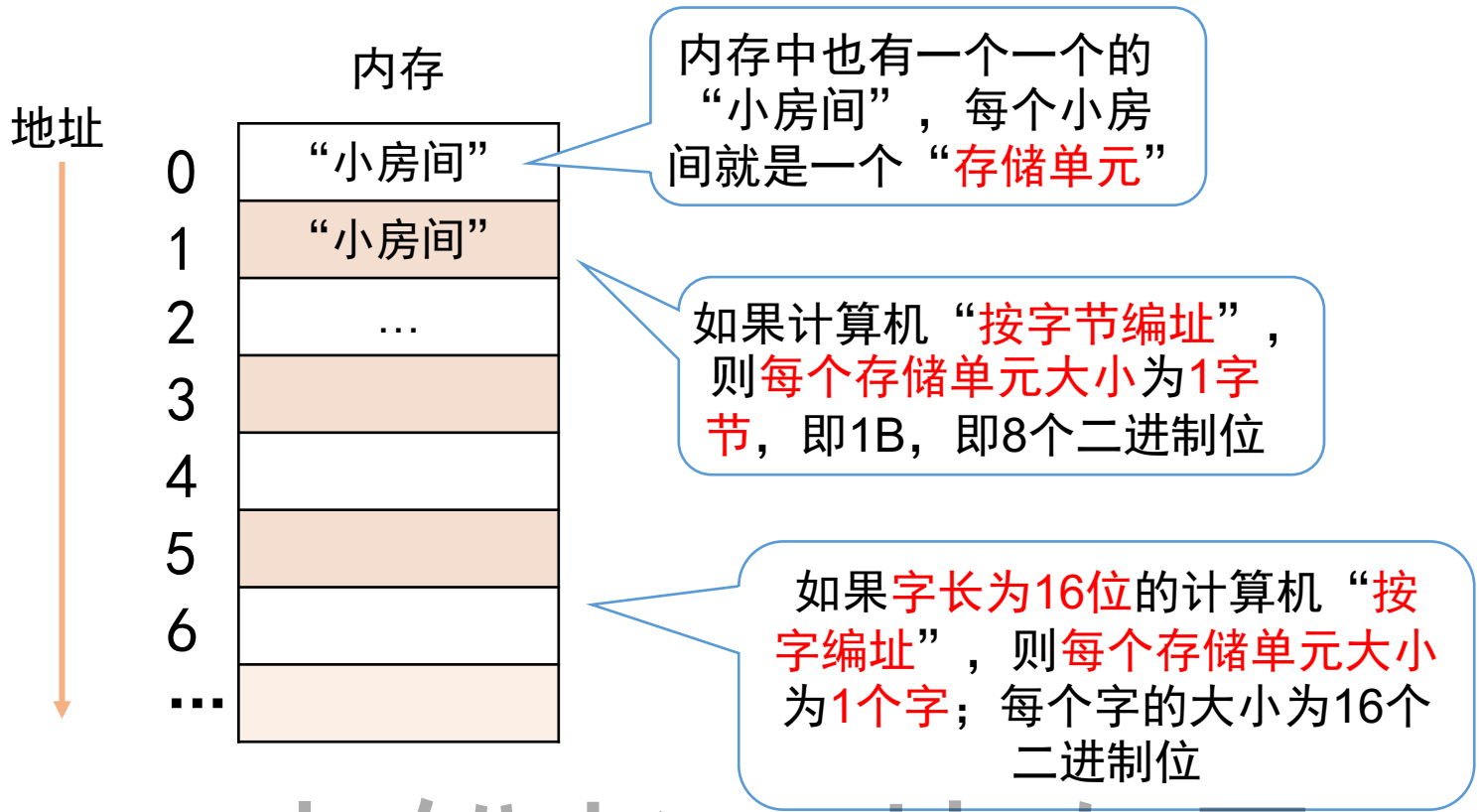
考研重点，必考知识点，求职重点

宿船长

内存管理的基本概念

内存管理的概念——什么是内存？

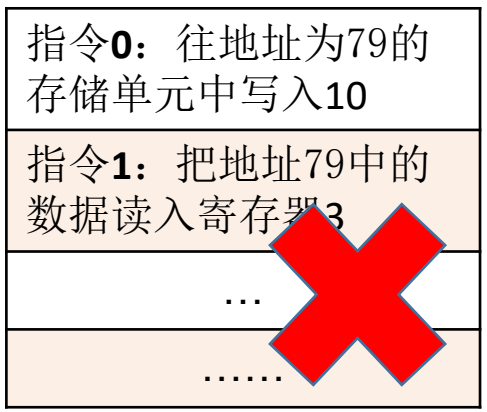
内存可存放数据。程序执行前需要先放到内存中才能被CPU处理——缓和CPU与硬盘之间的速度矛盾



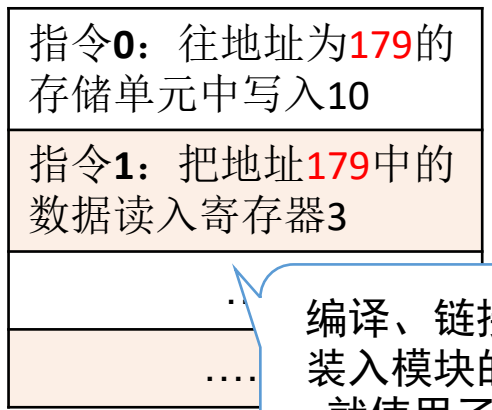
内存管理的概念——装入的三种方式（绝对装入）

绝对装入：在编译时，如果知道程序将放到内存中的哪个位置，编译程序将产生绝对地址的目标代码。装入程序按照装入模块中的地址，将程序和数据装入内存。

Eg：如果知道装入模块要从地址为100的地方开始存放…

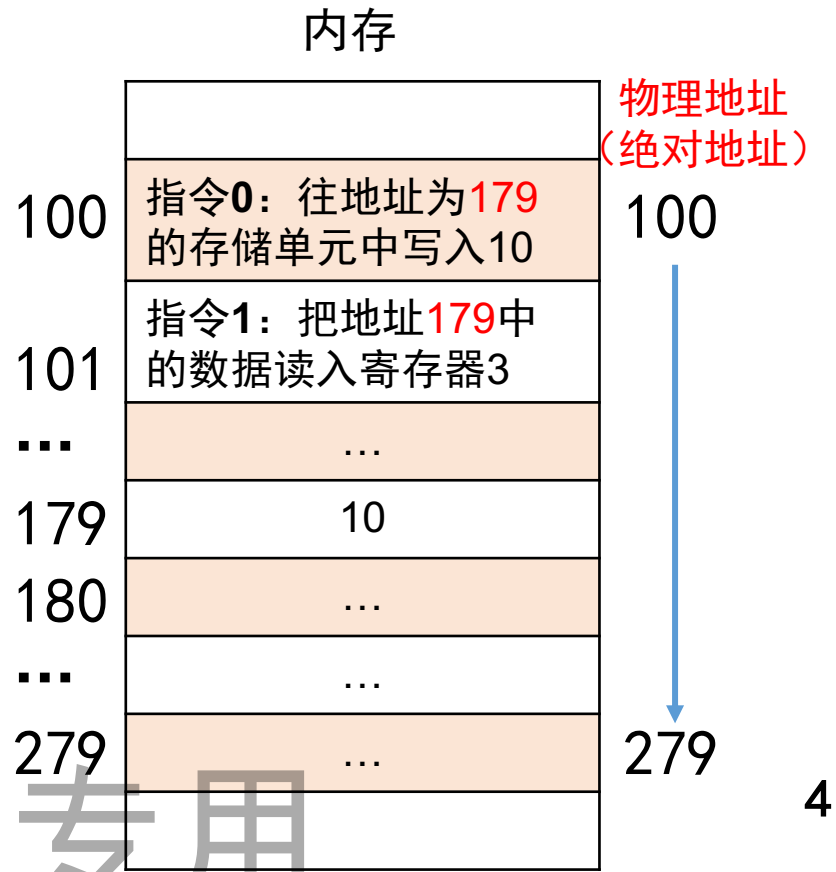


装入模块（可执行文件）



装入模块（可执行文件）

编译、链接后得到的装入模块的指令直接就使用了绝对地址

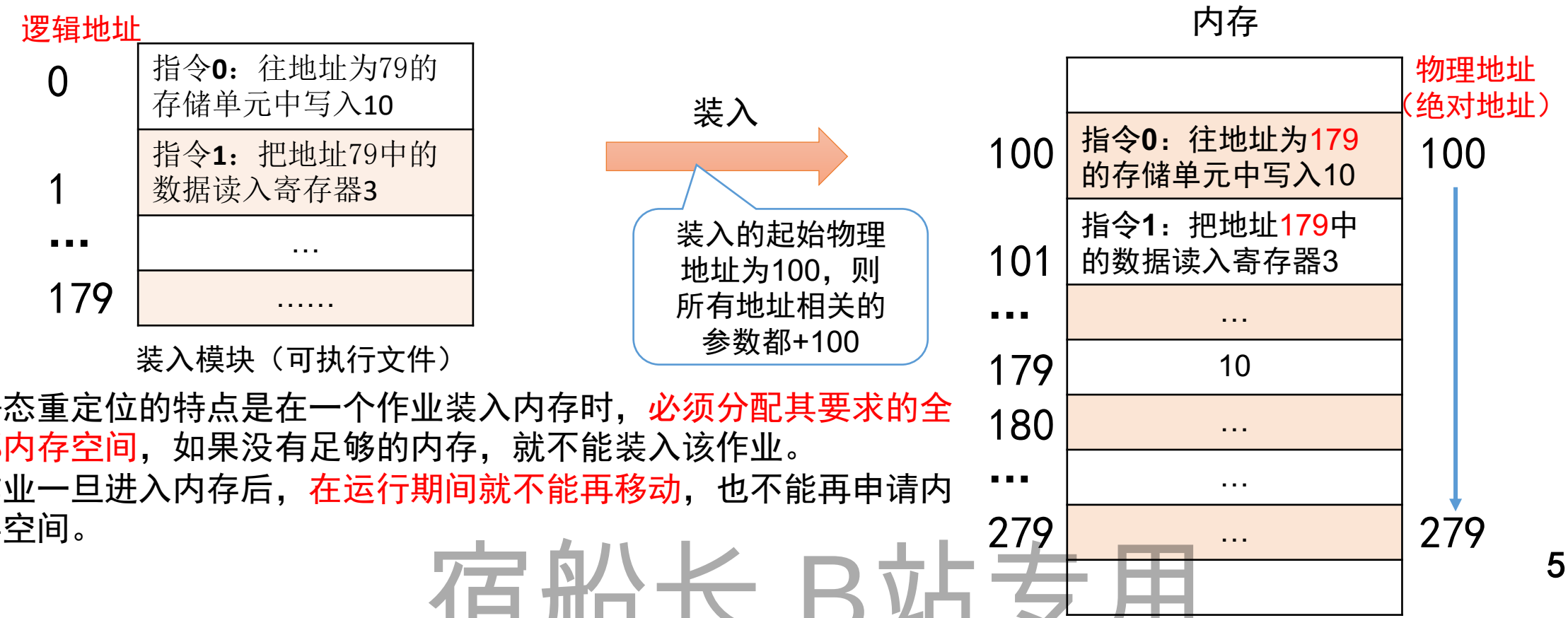


绝对装入只适用于单道程序环境。

程序中使用的绝对地址，可在编译或汇编时给出，也可由程序员直接赋予。通常情况下都是编译或汇编时再转换为绝对地址。

内存管理的概念——装入的三种方式（可重定位装入）

静态重定位：又称可重定位装入。编译、链接后的装入模块的地址都是从0开始的，指令中使用的地址、数据存放的地址都是相对于起始地址而言的逻辑地址。可根据内存的当前情况，将装入模块装入到内存的适当位置。装入时对地址进行“重定位”，将逻辑地址变换为物理地址（地址变换是在装入时一次完成的）

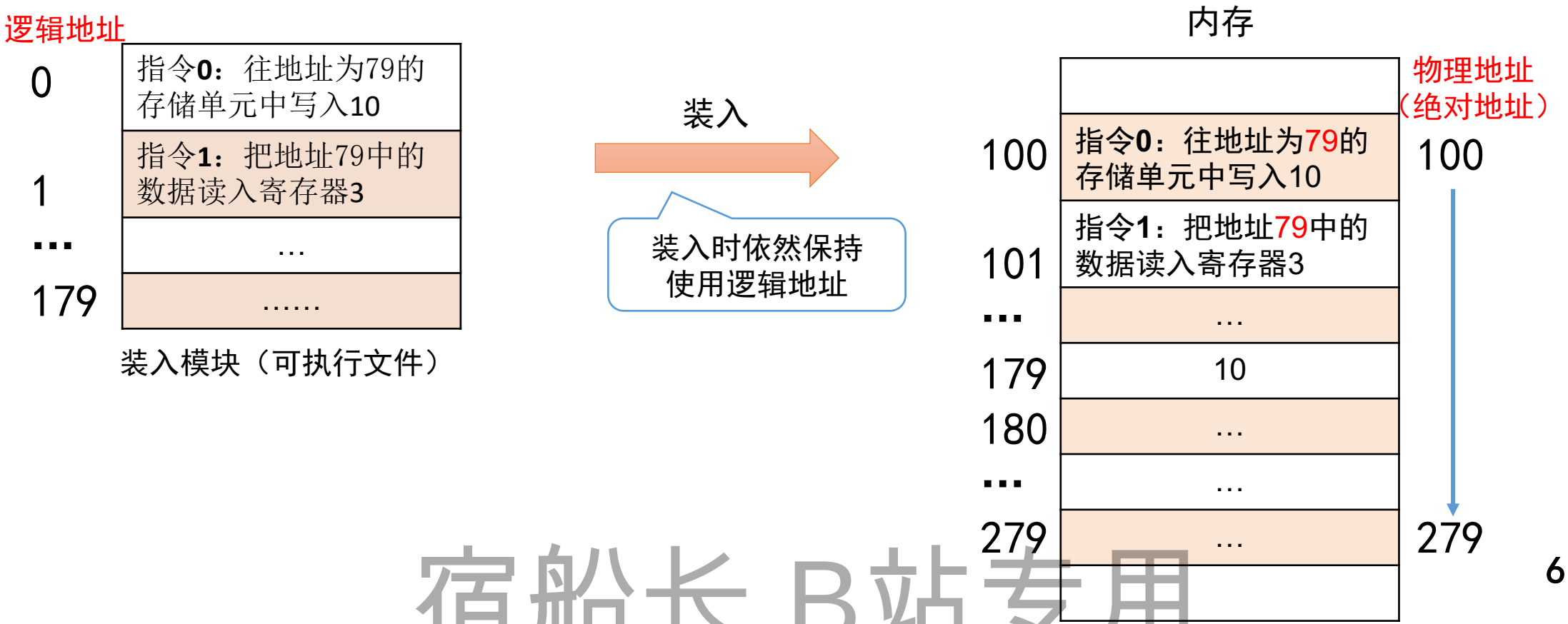


静态重定位的特点是在一个作业装入内存时，必须分配其要求的全部内存空间，如果没有足够的内存，就不能装入该作业。

作业一旦进入内存后，在运行期间就不能再移动，也不能再申请内存空间。

内存管理的概念——装入的三种方式（动态运行时装入）

动态重定位：又称动态运行时装入。编译、链接后的装入模块的地址都是从0开始的。装入程序把装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要执行时才进行。因此装入内存后所有的地址依然是逻辑地址。这种方式需要一个重定位寄存器的支持。

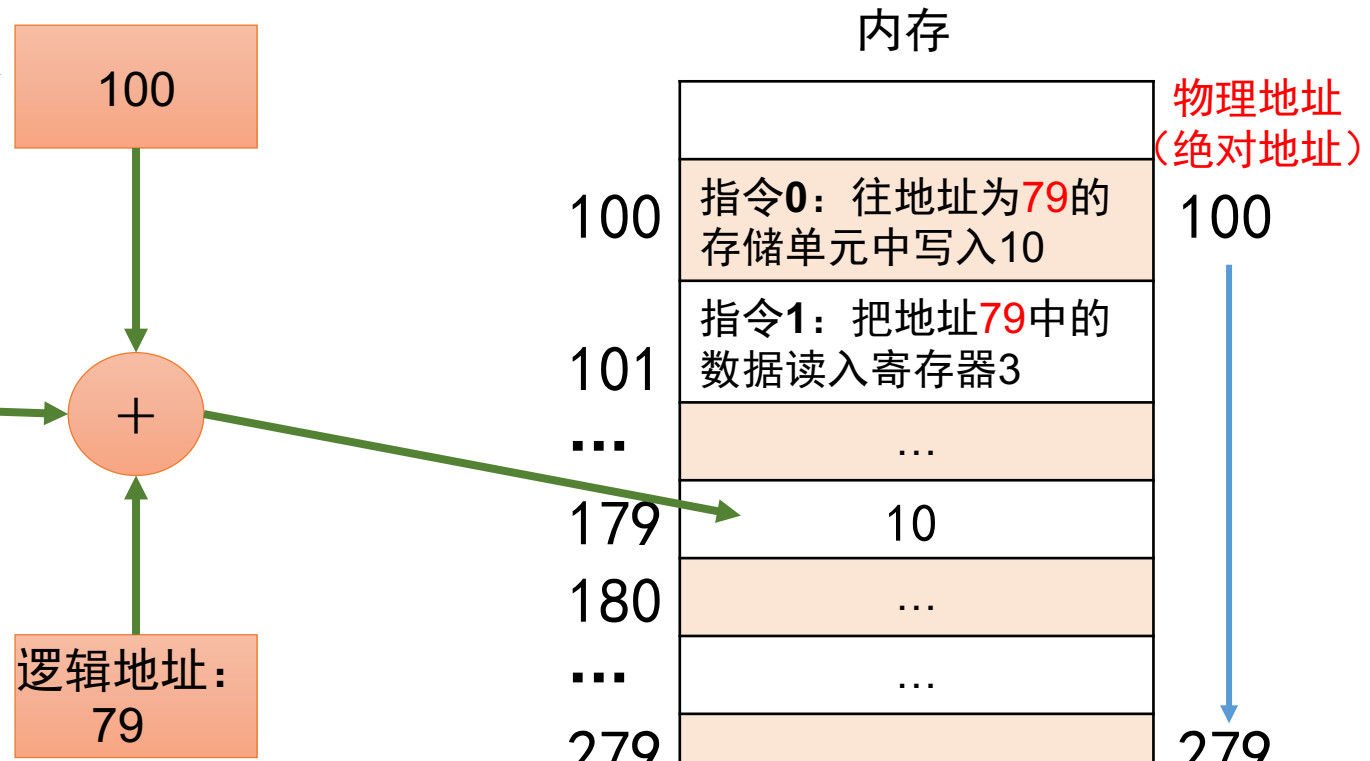


内存管理的概念——装入的三种方式（动态重定位）

动态重定位：又称动态运行时装入。编译、链接后的装入模块的地址都是从0开始的。装入程序把装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要执行时才进行。因此装入内存后所有的地址依然是逻辑地址。这种方式需要一个重定位寄存器的支持。

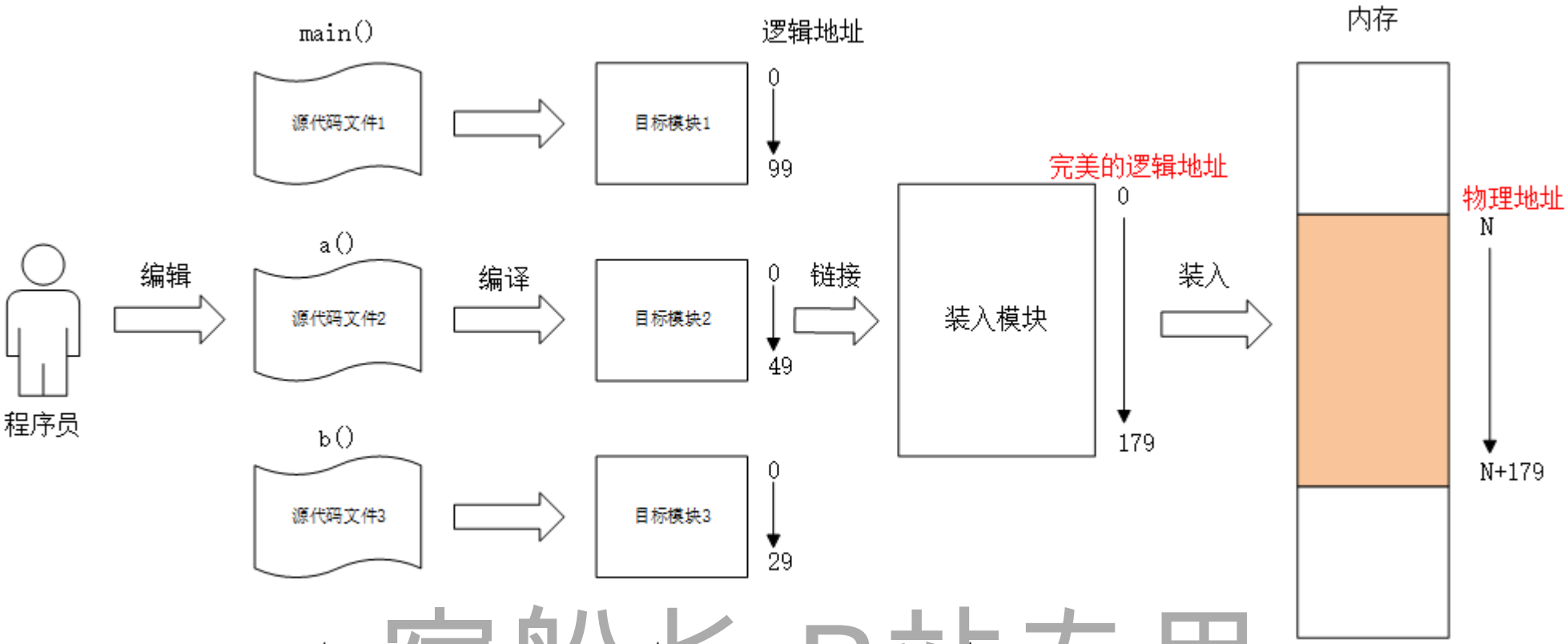
并且可将程序分配到不连续的存储区中；在程序运行前只需装入它的部分代码即可投入运行，然后在程序运行期间，根据需要动态申请分配内存；便于程序段的共享，可以向用户提供一个比存储空间大得多的地址空间。

采用动态重定位时允许程序在内存中发生移动。

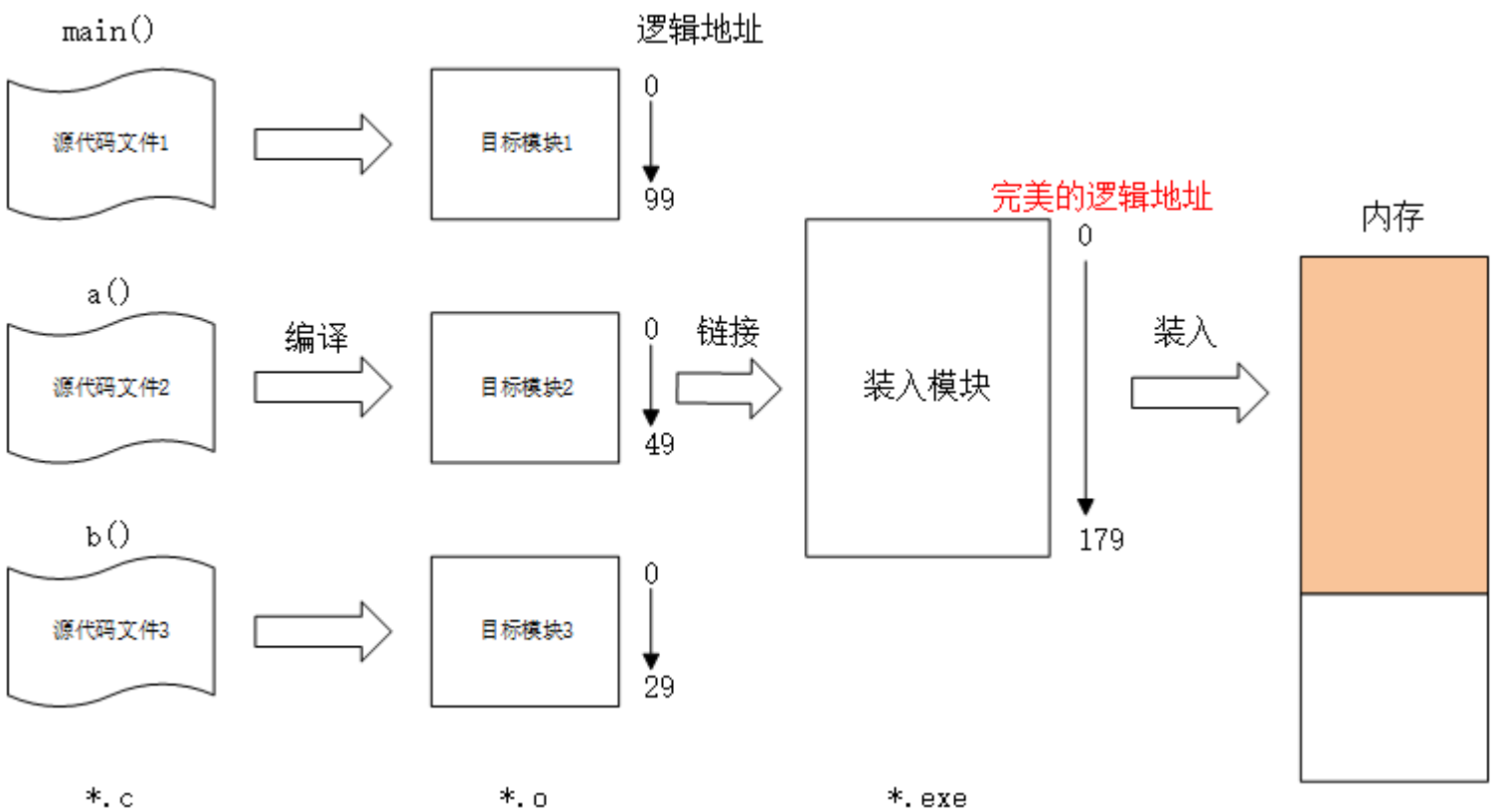


内存管理的概念——从写程序到程序运行

编译：由编译程序将用户源代码编译成若干个目标模块（编译就是把高级语言翻译为机器语言）
链接：由链接程序将编译后形成的一组目标模块，以及所需库函数链接在一起，形成一个完整的装入模块
装入（装载）：由装入程序将装入模块装入内存运行



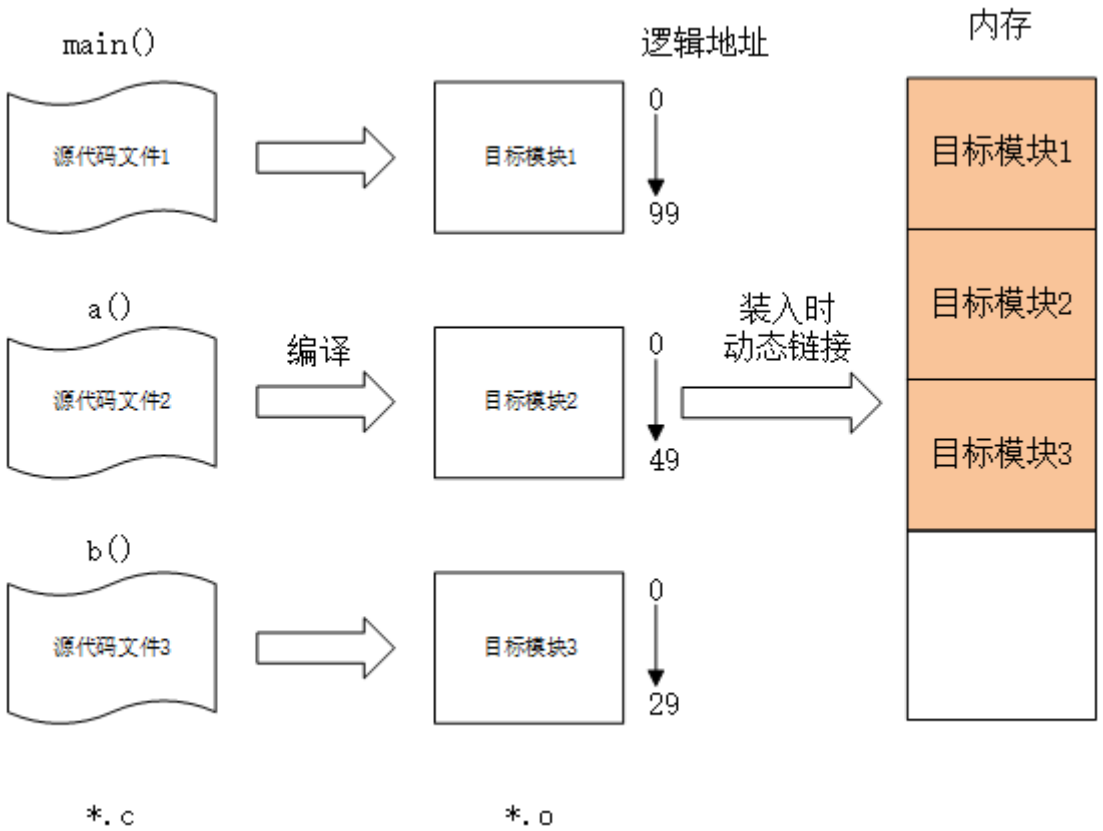
内存管理的概念——链接的三种方式



链接的三种方式：

1. 静态链接：在程序运行之前，先将各目标模块及它们所需的库函数连接成一个完整的可执行文件（装入模块），之后不再拆开。

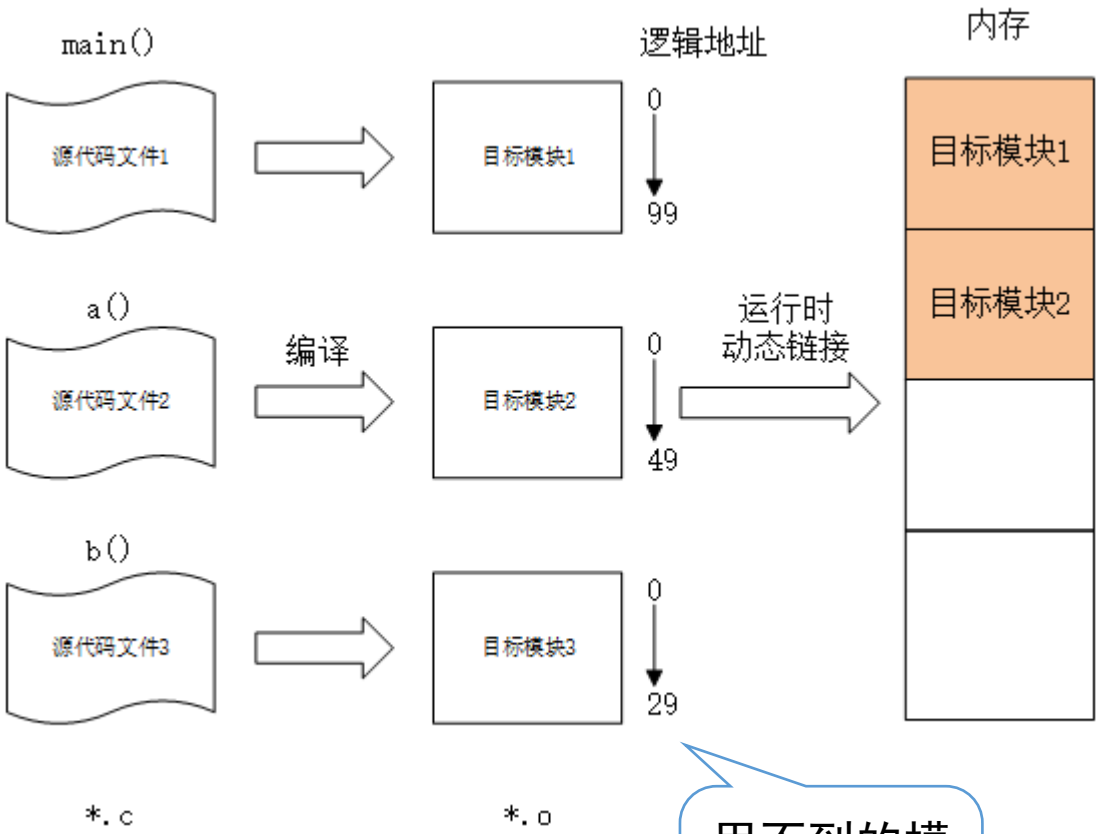
内存管理的概念——链接的三种方式



链接的三种方式：

1. 静态链接：在程序运行之前，先将各目标模块及它们所需的库函数连接成一个完整的可执行文件（装入模块），之后不再拆开。
2. 装入时动态链接：将各目标模块装入内存时，边装入边链接的链接方式。

内存管理的概念——链接的三种方式



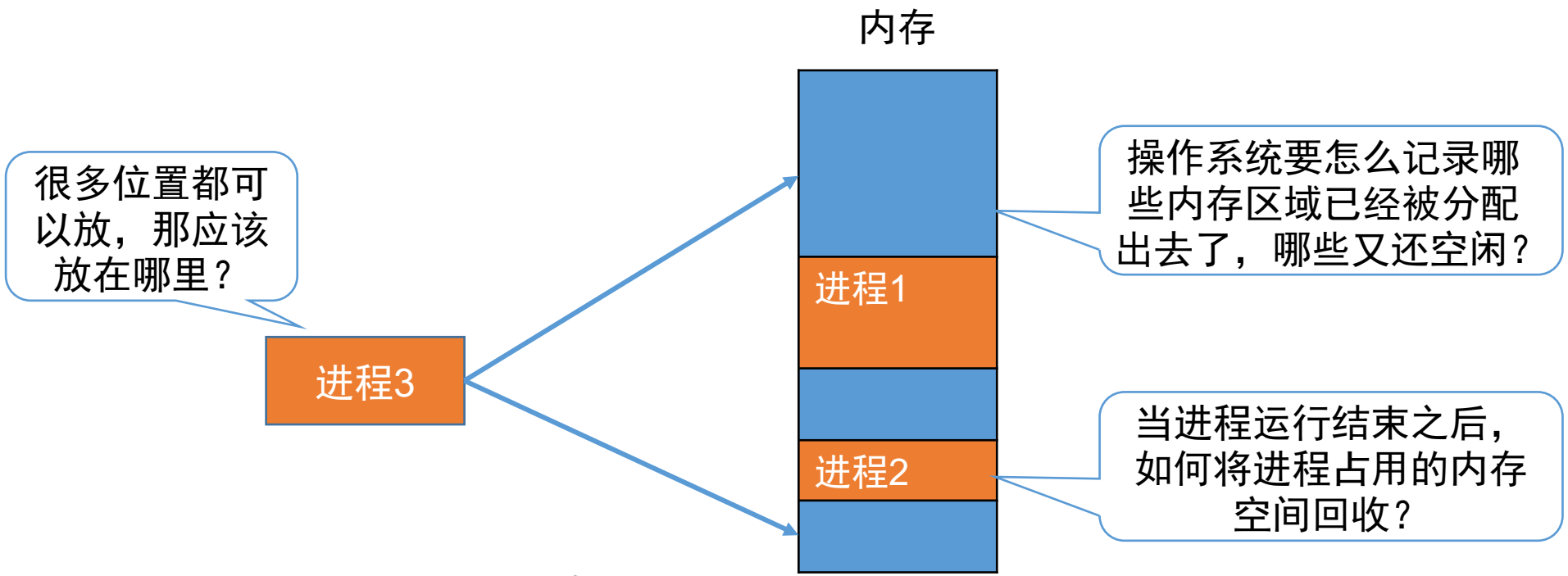
链接的三种方式：

1. 静态链接：在程序运行之前，先将各目标模块及它们所需的库函数连接成一个完整的可执行文件（装入模块），之后不再拆开。
2. 装入时动态链接：将各目标模块装入内存时，边装入边链接的链接方式。
3. 运行时动态链接：在程序执行中需要该目标模块时，才对它进行链接。其优点是便于修改和更新，便于实现对目标模块的共享。

内存管理的概念——内存空间的分配与回收

Q：操作系统作为系统资源的管理者，当然也需要对内存进行管理，要管些什么呢？

A1：操作系统负责***内存空间的分配与回收***



内存管理的概念——内存空间的扩展

Q：操作系统作为系统资源的管理者，当然也需要对内存进行管理，要管些什么呢？

A1：操作系统负责***内存空间的分配与回收***
A2：操作系统需要提供某种技术从逻辑上对***内存空间进行扩充***

游戏GTA的大小超过60GB，按理来说这个游戏程序运行之前需要把60GB数据全部放入内存。然而，实际的电脑内存可能才4GB，但为什么这个游戏可以顺利运行呢？
——虚拟技术（操作系统的虚拟性）

内存管理的概念——地址转换

Q：操作系统作为系统资源的管理者，当然也需要对内存进行管理，要管些什么呢？

A1：操作系统负责***内存空间的分配与回收***
A2：操作系统需要提供某种技术从逻辑上对***内存空间进行扩充***
A3：操作系统需要提供地址转换功能，负责程序的逻辑地址与物理地址的转换

为了使编程更方便，程序员写程序时应该只需要关注指令、数据的逻辑地址。而逻辑地址到物理地址的转换（这个过程称为地址重定位）应该由操作系统负责，这样就保证了程序员写程序时不需要关注物理内存的实际情况。

三种装入方式

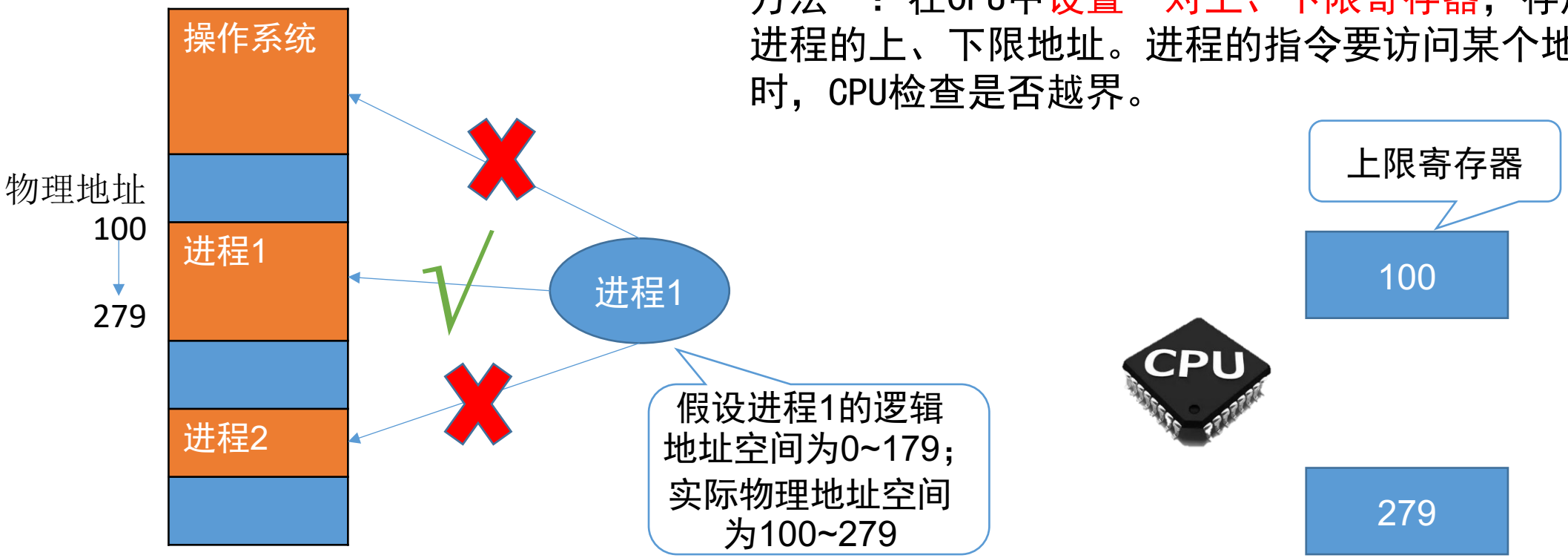
内存管理的概念——内存保护

Q：操作系统作为系统资源的管理者，当然也需要对内存进行管理，要管些什么呢？

A1：操作系统负责***内存空间的分配与回收***
A2：操作系统需要提供某种技术从逻辑上对***内存空间进行扩充***
A3：操作系统需要提供地址转换功能，负责程序的逻辑地址与物理地址的转换
A4：操作系统需要提供***内存保护***功能。保证各进程在各自存储空间内运行，互不干扰

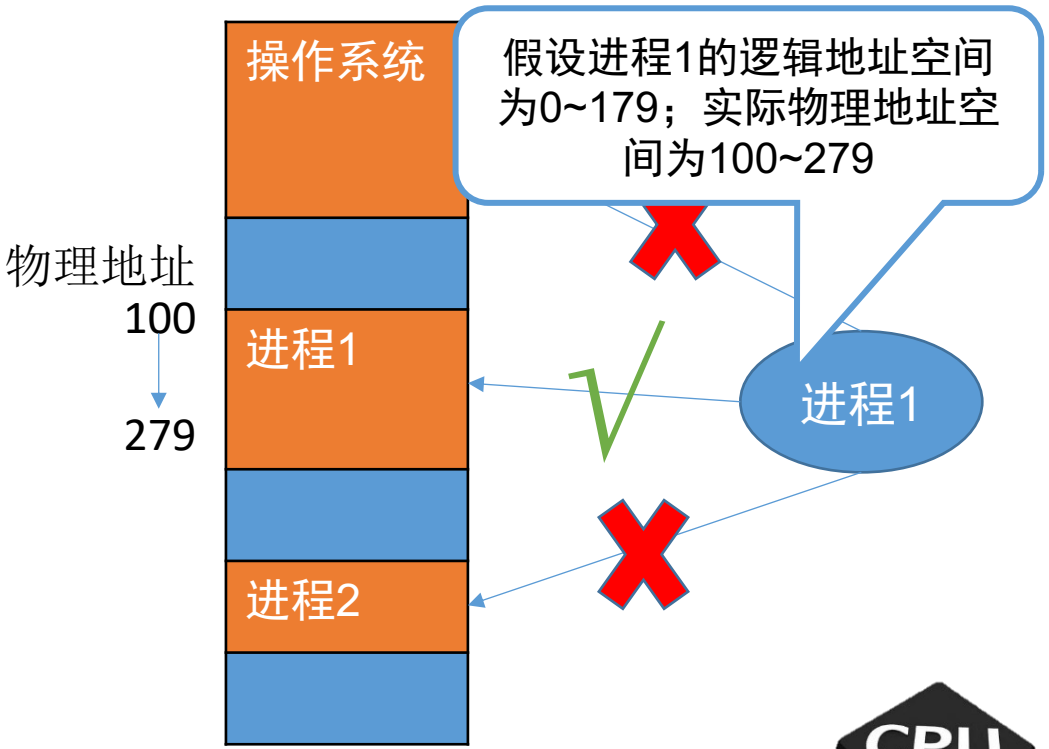
内存管理的概念——内存保护

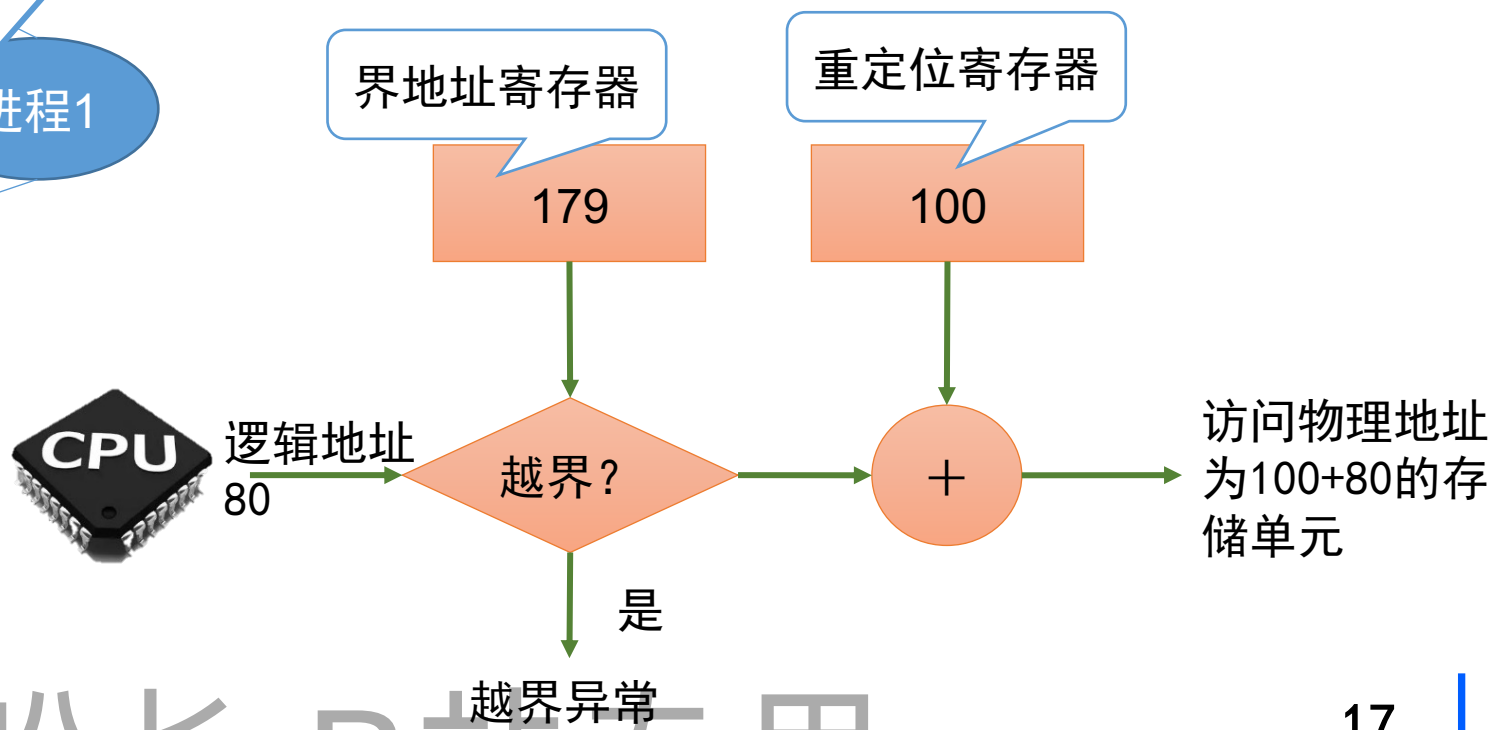
内存保护可采取两种方法：
方法一：在CPU中设置一对上、下限寄存器，存放进程的上、下限地址。进程的指令要访问某个地址时，CPU检查是否越界。



内存管理的概念——内存保护

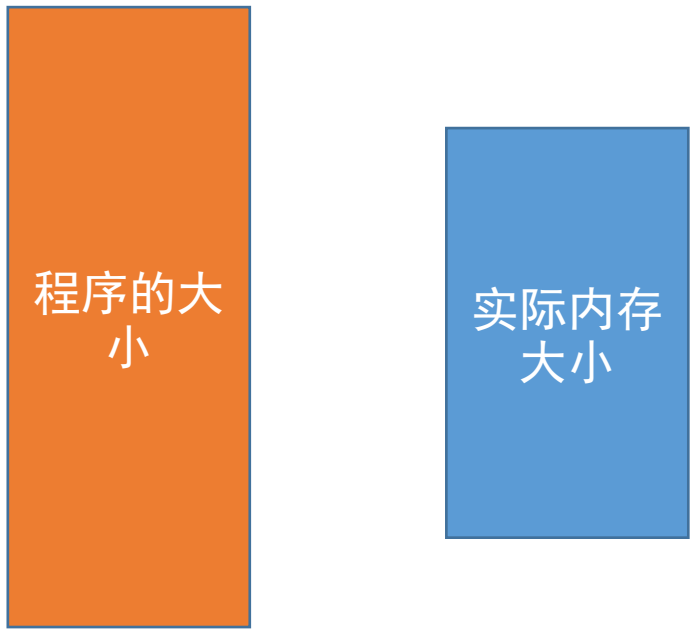
内存保护可采取两种方法：
方法二：采用重定位寄存器（又称基址寄存器）和界地址寄存器（又称限长寄存器）进行越界检查。重定位寄存器中存放的是进程的起始物理地址。界地址寄存器中存放的是进程的最大逻辑地址。





内存管理的概念——内存保护

早期的计算机内存很小，比如IBM推出的第一台PC机最大只支持1MB大小的内存。因此经常会出现内存大小不够的情况。
后来人们引入了覆盖技术，用来解决“程序大小超过物理内存总和”的问题

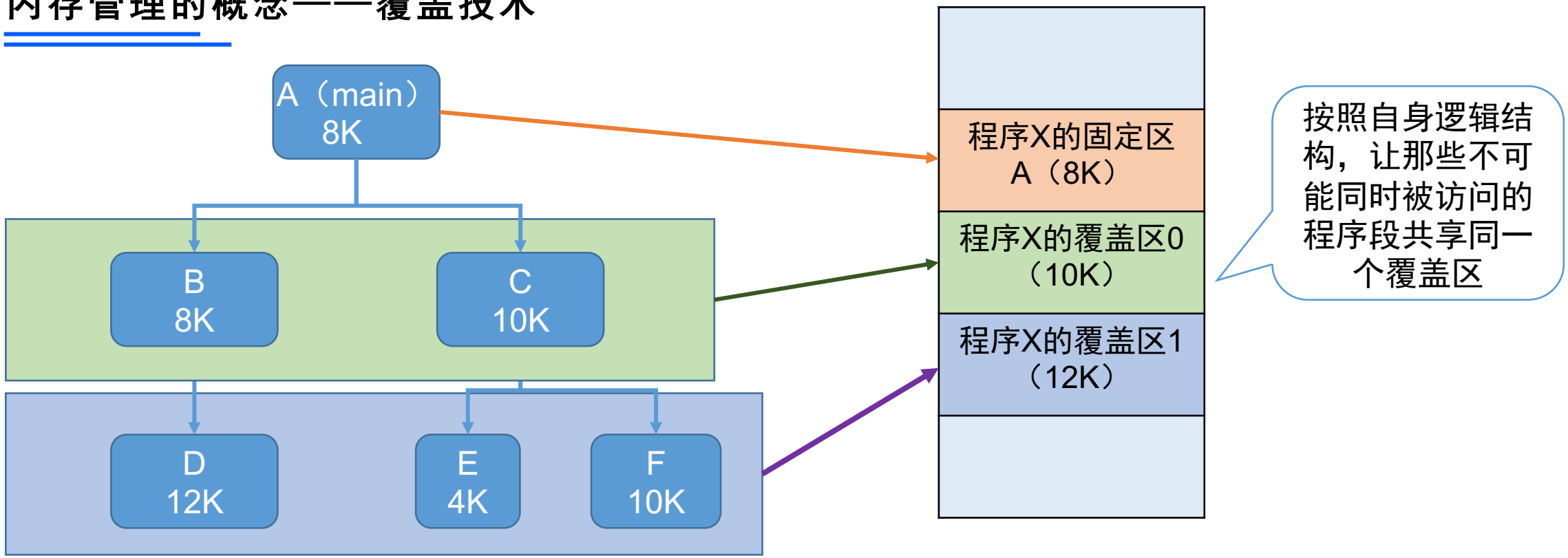


覆盖技术的思想：将程序分为多个段（多个模块）。
常用的段常驻内存，不常用的段在需要时调入内存。

内存中分为一个“固定区”和若干个“覆盖区”。

需要常驻内存的段放在“固定区”中，调入后就不再调出（除非运行结束）
不常用的段放在“覆盖区”，需要用到时调入内存，用不到时调出内存

内存管理的概念——覆盖技术

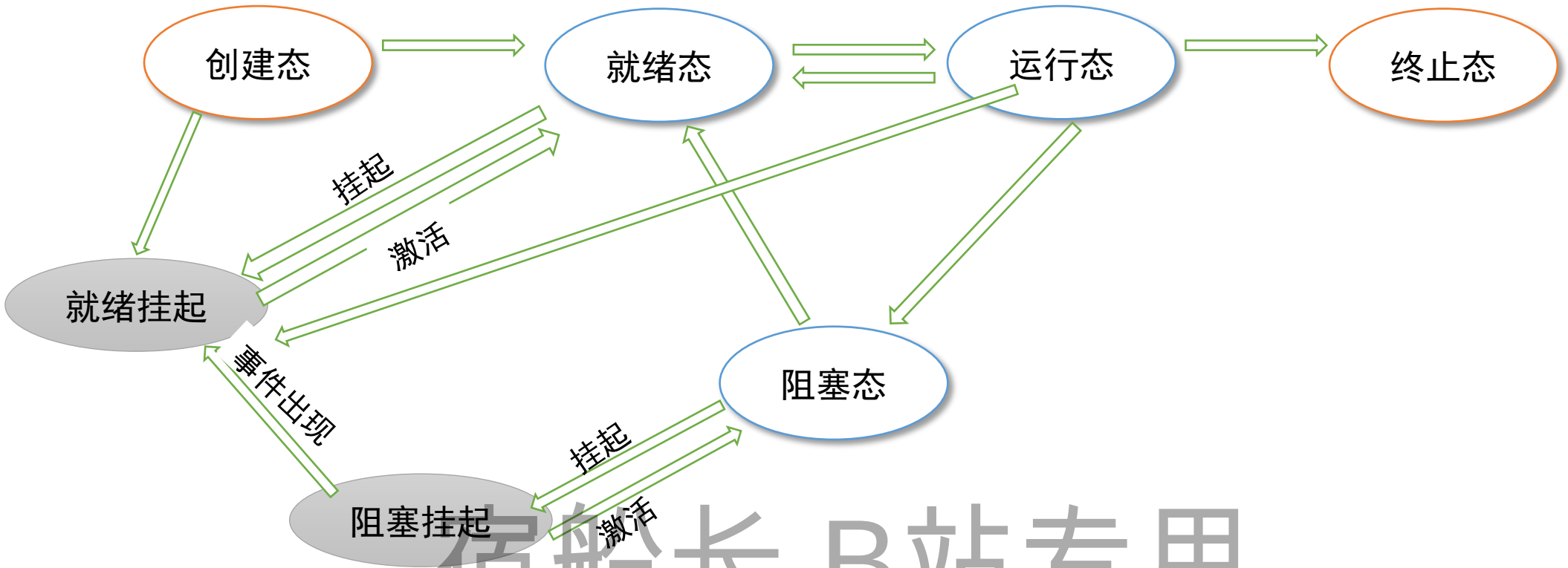


必须由程序员声明覆盖结构，操作系统完成自动覆盖。缺点：对用户不透明，增加了用户编程负担。
覆盖技术只用于早期的操作系统中，现在已成为历史。

内存管理的概念——交换技术

交换（对换）技术的设计思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存（进程在内存与磁盘间动态调度）

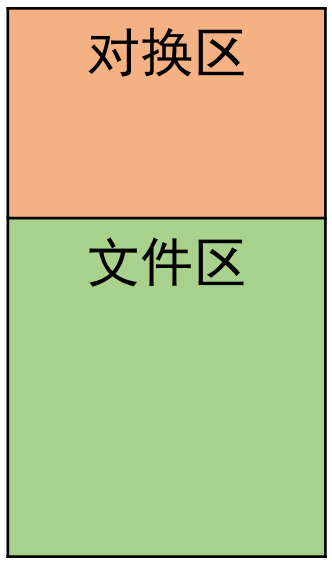
暂时换出外存等待的进程状态为挂起状态（挂起态，suspend）
挂起态又可以进一步细分为就绪挂起、阻塞挂起两种状态



内存管理的概念——交换技术

交换（对换）技术的设计思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存（进程在内存与磁盘间动态调度）

1. 应该在外存（磁盘）的什么位置保存被换出的进程？
2. 什么时候应该交换？
3. 应该换出哪些进程？



磁盘存储空间

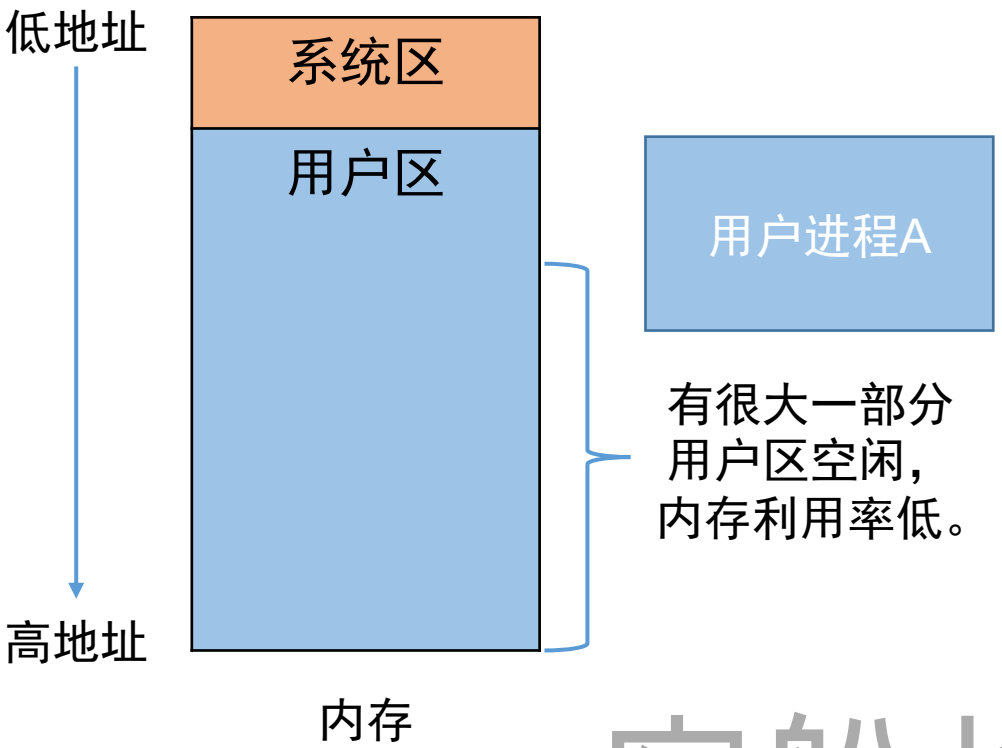
1. 具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式；对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式（学过文件管理章节后即可理解）。总之，对换区的I/O速度比文件区的更快。
2. 交换通常在许多进程运行且内存吃紧时进行，而系统负荷降低就暂停。例如：在发现许多进程运行时经常发生缺页，就说明内存紧张，此时可以换出一些进程；如果缺页率明显下降，就可以暂停换出。
3. 可优先换出阻塞进程；可换出优先级低的进程；为了防止优先级低的进程在被调入内存后很快又被换出，有的系统还会考虑进程在内存的驻留时间…

（注意：PCB会常驻内存，不会被换出外存）

内存分配方式

内存分配方式——连续分配管理方式（单一连续分配）

在单一连续分配方式中，内存被分为系统区和用户区。
系统区通常位于内存的低地址部分，用于存放操作系统相关数据；用户区用于存放用户进程相关数据。
内存中只能有一道用户程序，用户程序独占整个用户区空间。



优点：实现简单；无外部碎片；可以采用覆盖技术扩充内存；不一定需要采取内存保护（eg：早期的PC操作系统MS-DOS）。
缺点：只能用于单用户、单任务的操作系统中；有内部碎片；存储器利用率极低。

分配给某进程的内存区域中，如果有些部分没有用上，就是“内部碎片”

内存分配方式——连续分配管理方式（固定分区分配）

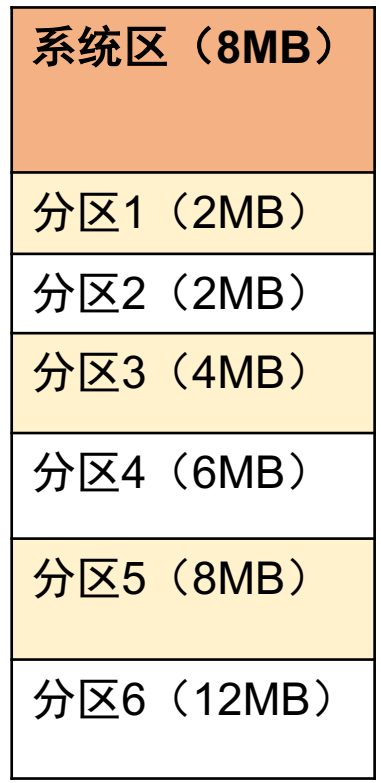
20世纪60年代出现了支持多道程序的系统，为了能在内存中装入多道程序，且这些程序之间又不会相互干扰，于是将整个用户空间划分为若干个固定大小的分区，在每个分区中只装入一道作业，这样就形成了最早的、最简单的一种可运行多道程序的内存管理方式。

固定分区分配
- 分区大小相等
- 分区大小不等

分区大小相等：缺乏灵活性，但是很适合用于用一台计算机控制多个相同对象的场合（比如：钢铁厂有n个相同的炼钢炉，就可把内存分为n个大小相等的区域存放n个炼钢炉控制程序）

分区大小不等：增加了灵活性，可以满足不同大小的进程需求。根据常在系统中运行的作业大小情况进行划分（比如：划分多个小分区、适量中等分区、少量大分区）

内存（分区大小相等）



内存（分区大小不等）

内存分配方式——连续分配管理方式（固定分区分配）

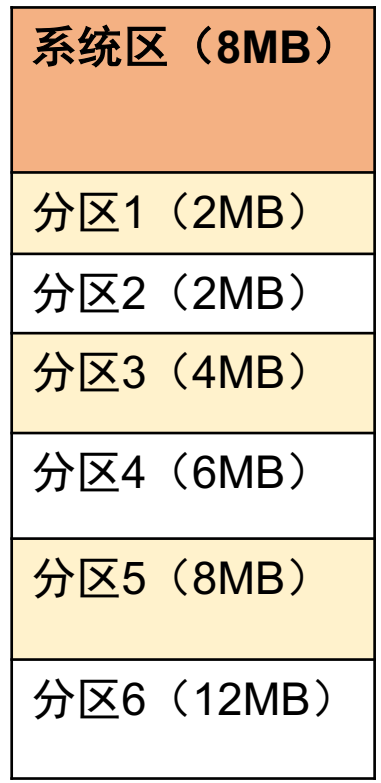
操作系统需要建立一个数据结构——分区说明表，来实现各个分区的分配与回收。每个表项对应一个分区，通常按分区大小排列。每个表项包括对应分区的大小、起始地址、状态（是否已分配）。

分区号	大小（MB）	起始地址（M）	状态
1	2	8	未分配
2	2	10	未分配
3	4	12	已分配
……	……	……	……

用数据结构的数组（或链表）即可表示这个表

当某用户程序要装入内存时，由操作系统内核程序根据用户程序大小检索该表，从中找到一个能满足大小的、未分配的分区，将之分配给该程序，然后修改状态为“已分配”。

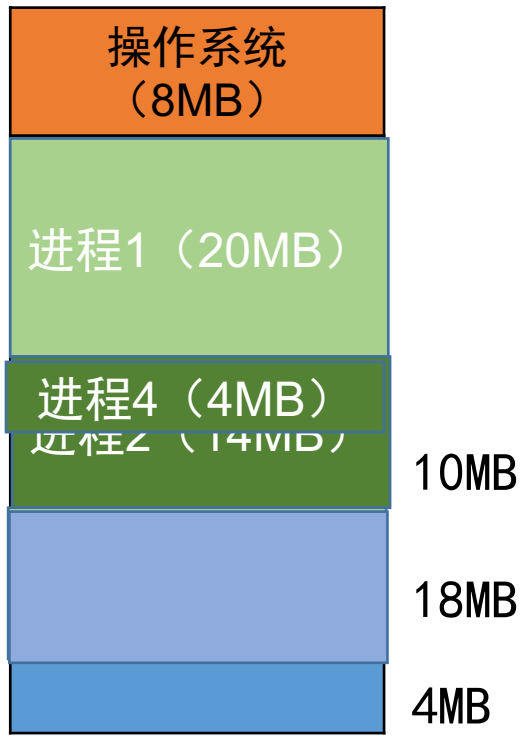
优点：实现简单，无外部碎片。
缺点：a. 当用户程序太大时，可能所有的分区都不能满足需求，此时不得不采用覆盖技术来解决，但这又会降低性能；b. 会产生内部碎片，内存利用率低。



内存（分区大小不等）

内存分配方式——连续分配管理方式（动态分区分配）

动态分区分配又称为可变分区分配。这种分配方式不会预先划分内存分区，而是在进程装入内存时，根据进程的大小动态地建立分区，并使分区的大小正好适合进程的需要。因此系统分区的大小和数目是可变的。
（eg：假设某计算机内存大小为64MB，系统区8MB，用户区共56 MB…）



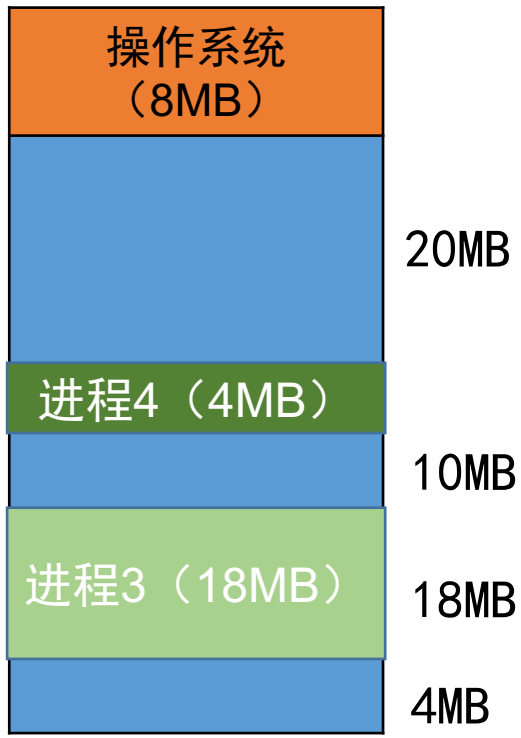
Q1：系统要用什么样的数据结构记录内存的使用情况？
Q2：当很多个空闲分区都能满足需求时，应该选择哪个分区进行分配？
Q3：如何进行分区的分配与回收操作？

内存分配方式——连续分配管理方式（动态分区分配）

Q1：系统要用什么样的数据结构记录内存的使用情况？

A1：

两种常用的数据结构：（1）空闲分区表；（2）空闲分区链



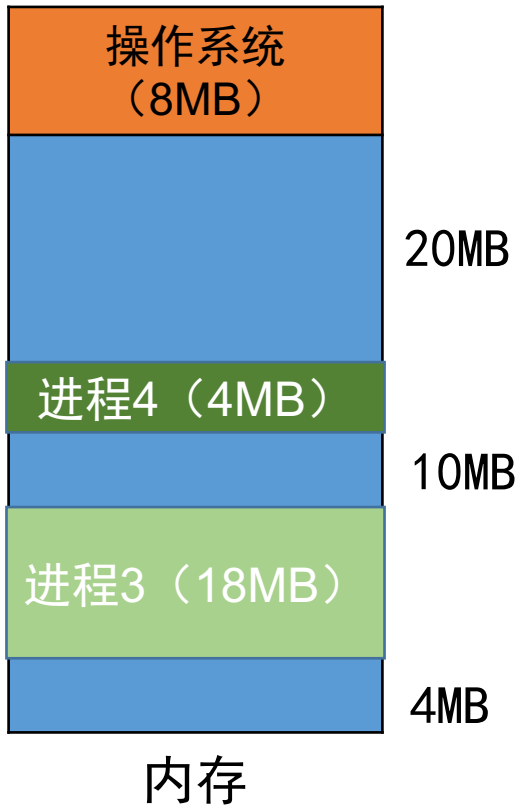
分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

空闲分区表：每个空闲分区对应一个表项。表项中包含分区号、分区大小分区起始地址等信息

空闲分区链：每个分区的起始部分和末尾部分分别设置前向指针和后向指针。起始部分处还可记录分区大小等信息

内存分配方式——连续分配管理方式（动态分区分配）

Q2：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？



进程5（4MB）

应该用最大的分区进行分配？还是用最小的分区进行分配？又或是用地址最低的部分进行分配？

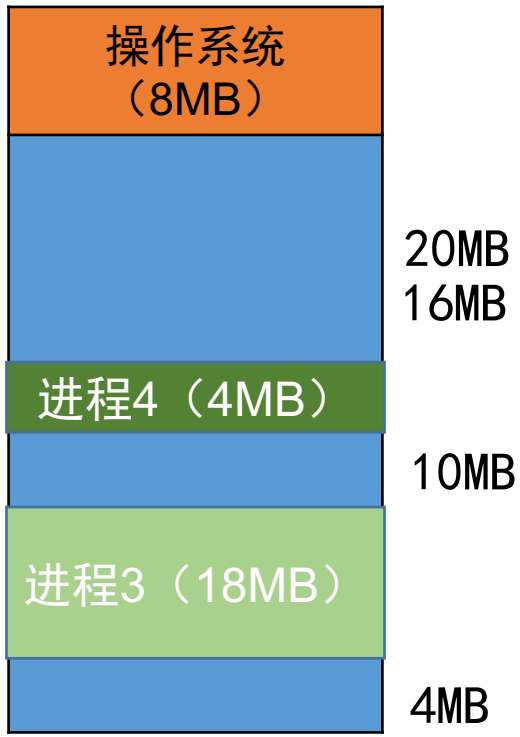
把一个新作业装入内存时，须按照一定的动态分区分配算法，从空闲分区表（或空闲分区链）中选出一个分区分配给该作业。由于分配算法算法对系统性能有很大的影响，因此人们对它进行了广泛的研究。

下个小节会介绍四种动态分区分配算法…

内存分配方式——连续分配管理方式（动态分区分配）

Q3：如何进行分区的分配与回收操作？

A1：假设系统采用的数据结构是“空闲分区表”...如何分配？



进程5（4MB）

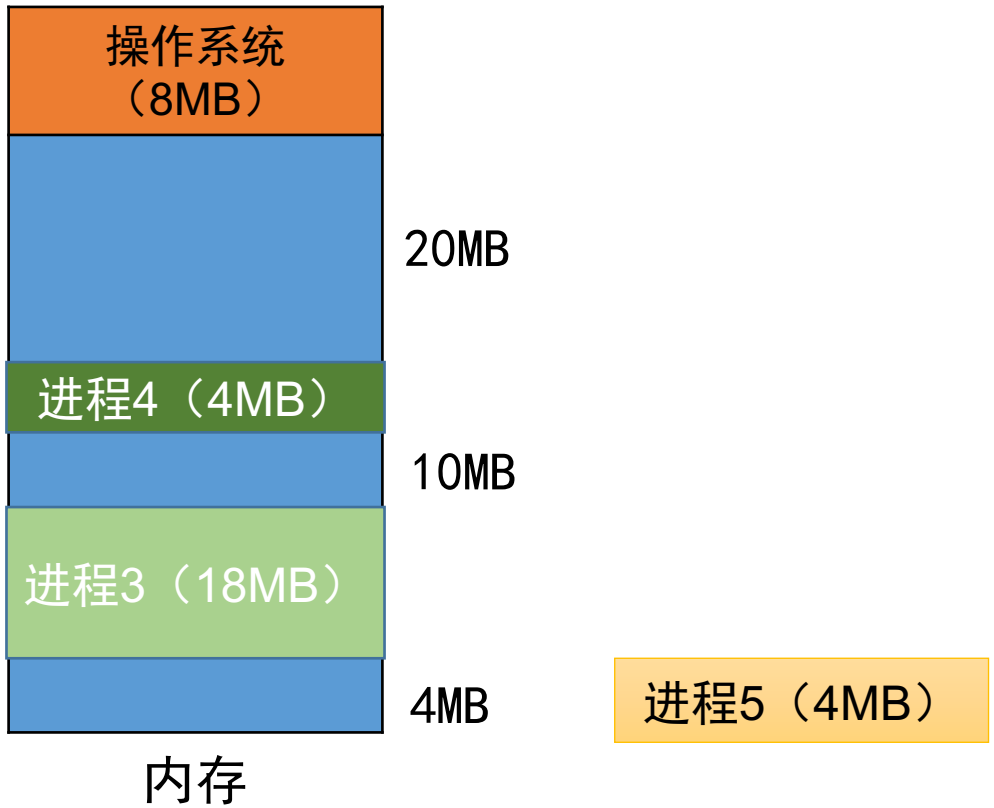
分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

分区号	分区大小（MB）	起始地址（M）	状态
1	16	12	空闲
2	10	32	空闲
3	4	60	空闲

内存分配方式——连续分配管理方式（动态分区分配）

Q3：如何进行分区的分配与回收操作？

A1：假设系统采用的数据结构是“空闲分区表”...如何分配？



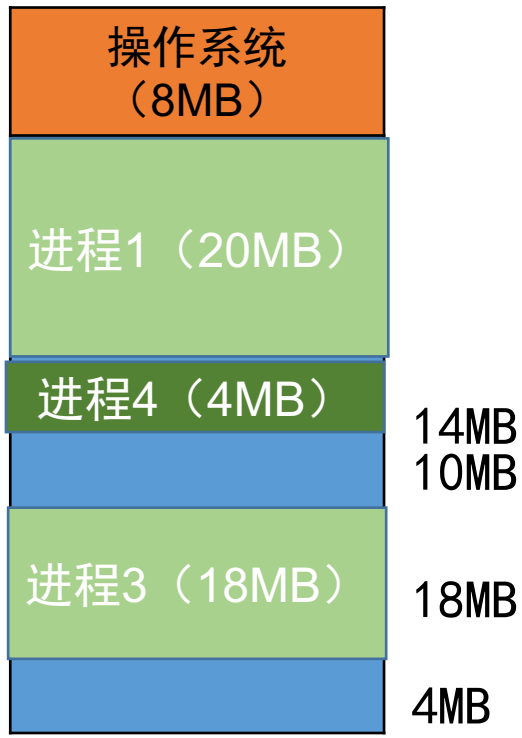
分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲

内存分配方式——连续分配管理方式（动态分区分配）

Q3：如何进行分区的分配与回收操作？

A1：假设系统采用的数据结构是“空闲分区表”...如何分配？



情况一：回收区的后面有一个相邻的空闲分区

分区号	分区大小（MB）	起始地址（M）	状态
1	10	32	空闲
2	4	60	空闲

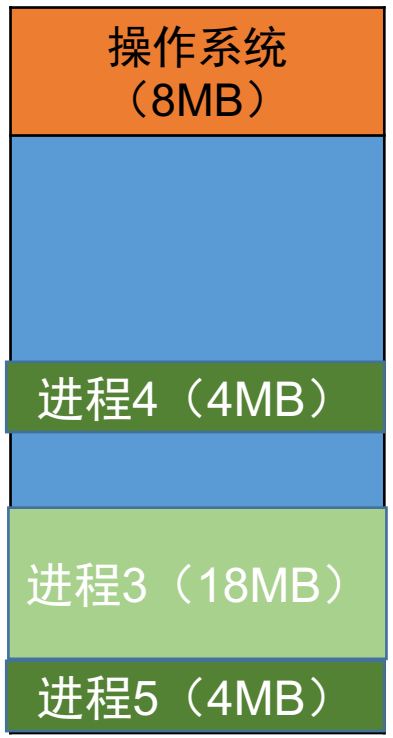
分区号	分区大小（MB）	起始地址（M）	状态
1	14	28	空闲
2	4	60	空闲

两个相邻的空闲分区合并为一个

内存分配方式——连续分配管理方式（动态分区分配）

Q3：如何进行分区的分配与回收操作？

A1：假设系统采用的数据结构是“空闲分区表”...如何分配？



10MB

28MB

内存

情况二：回收区的前面有一个相邻的空闲分区

分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲

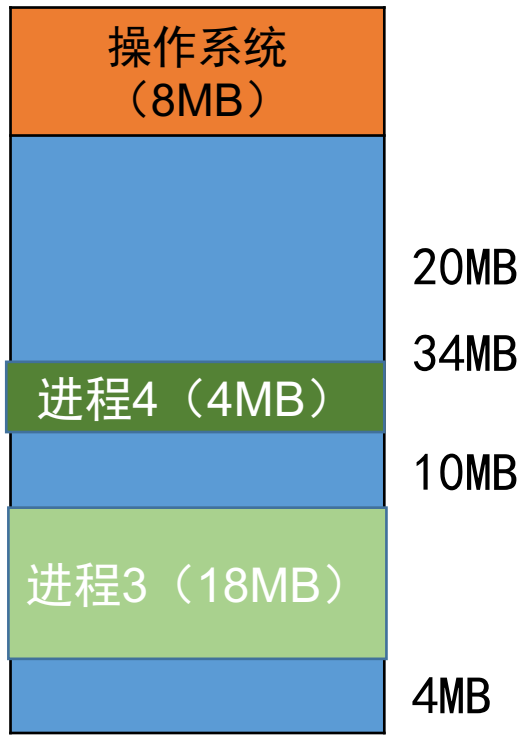
分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	28	60	空闲

两个相邻的空闲分区合并为一个

内存分配方式——连续分配管理方式（动态分区分配）

Q3：如何进行分区的分配与回收操作？

A1：假设系统采用的数据结构是“空闲分区表”...如何分配？



内存

情况三：回收区的前、后各有一个相邻的空闲分区

分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

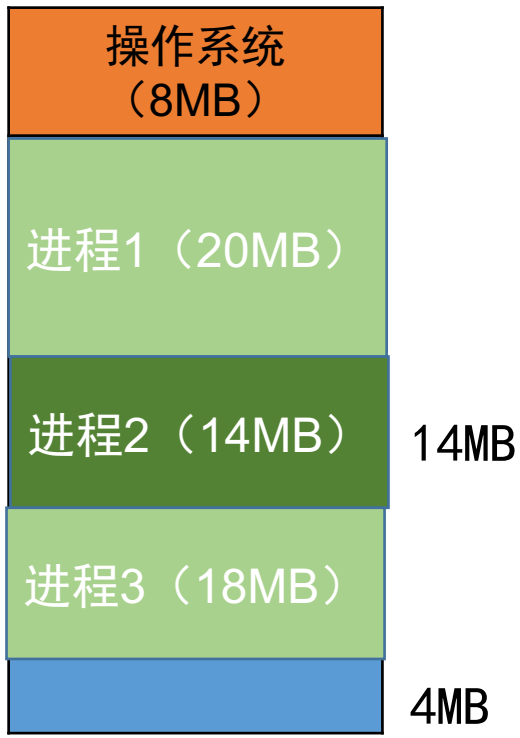
分区号	分区大小（MB）	起始地址（M）	状态
1	34	8	空闲
2	4	60	空闲

三个相邻的空闲分区合并为一个

内存分配方式——连续分配管理方式（动态分区分配）

Q3：如何进行分区的分配与回收操作？

A1：假设系统采用的数据结构是“空闲分区表”...如何分配？



情况四：回收区的前、后都没有相邻的空闲分区

分区号	分区大小（MB）	起始地址（M）	状态
1	4	60	空闲

分区号	分区大小（MB）	起始地址（M）	状态
1	14	28	空闲
2	4	60	空闲

新增一个表项

注：各表项的顺序不一定按照地址递增顺序排列，具体的排列方式需要依据动态分区分配算法来确定。

内存分配方式——连续分配管理方式（动态分区分配）

动态分区分配又称为可变分区分配。这种分配方式不会预先划分内存分区，而是在进程装入内存时，根据进程的大小动态地建立分区，并使分区的大小正好适合进程的需要。因此系统分区的大小和数目是可变的。

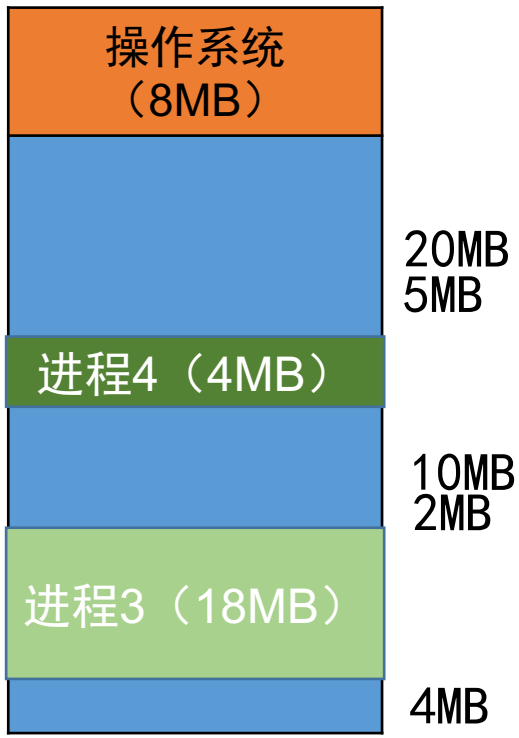
动态分区分配没有内部碎片，但是有外部碎片。
内部碎片，分配给某进程的内存区域中，如果有些部分没有用上。
外部碎片，是指内存中的某些空闲分区由于太小而难以利用。

如果内存中空闲空间的总和本来可以满足某进程的要求，但由于进程需要的是一整块连续的内存空间，因此这些“碎片”不能满足进程的需求。
可以通过紧凑（拼凑，Compaction）技术来解决外部碎片。

内存分配方式——动态分区分配算法（首次适应算法）

算法思想：每次都从低地址开始查找，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。



进程5（15MB）

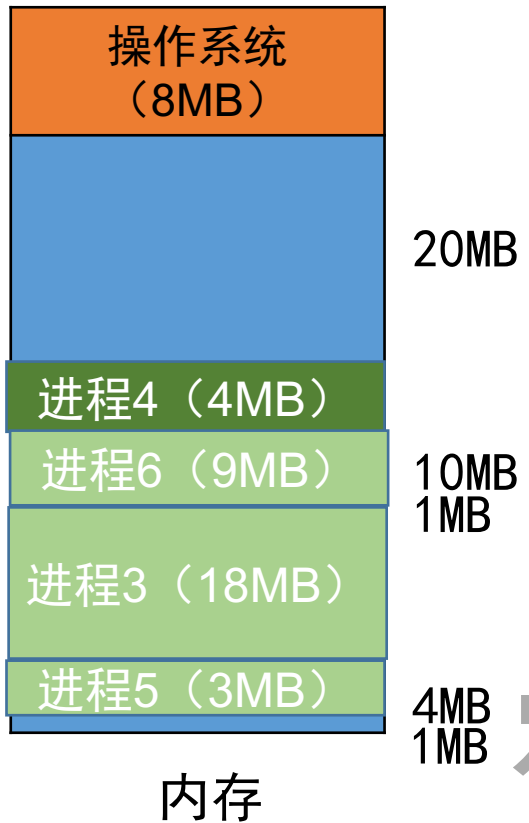
进程6（8MB）

分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

内存分配方式——动态分区分配算法（最佳适应算法）

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。



分区号	分区大小（MB）	起始地址（M）	状态
1	4	60	空闲
2	10	32	空闲
3	20	8	空闲

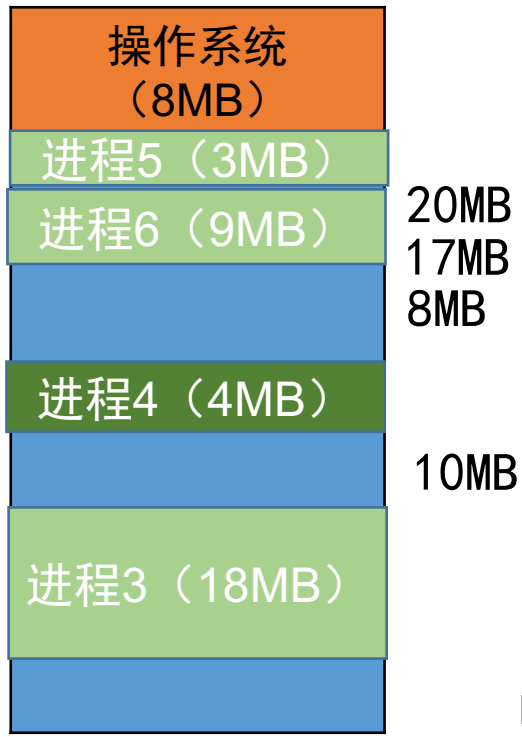
缺点：每次都选最小的分区进行分配，会留下越来越多的、很小的、难以利用的内存块。因此这种方法会产生很多的外部碎片。

内存分配方式——动态分区分配算法（最坏适应算法）

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——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。



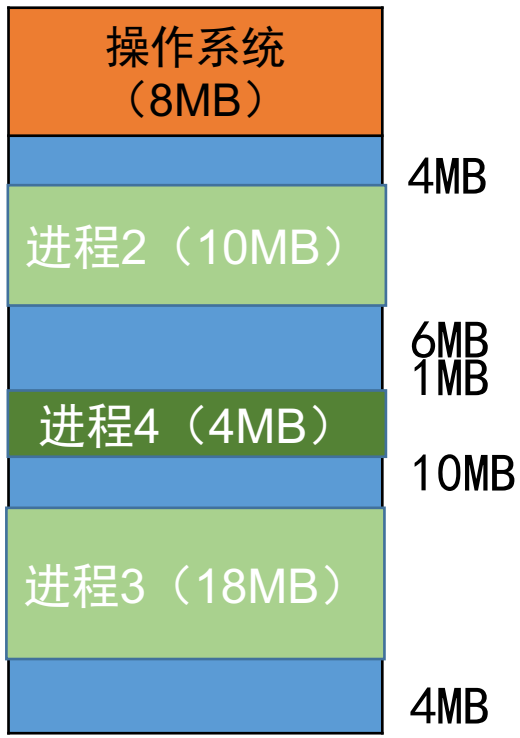
分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

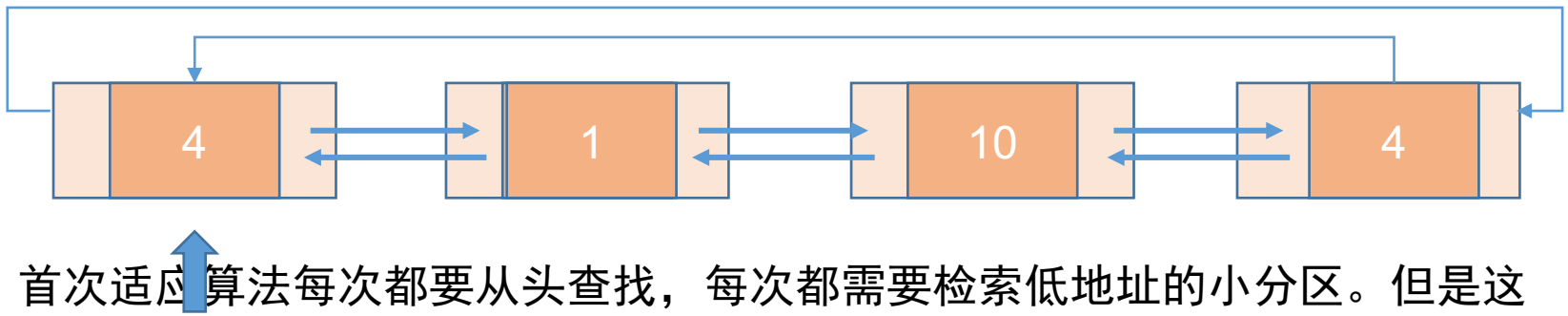
缺点：每次都选最大的分区进行分配，虽然可以让分配后留下的空闲区更大，更可用，但是这种方式会导致较大的连续空闲区被迅速用完。如果之后有“大进程”到达，就没有内存分区可用了。

内存分配方式——动态分区分配算法（邻近适应算法）

算法思想：首次适应算法每次都从链头开始查找的。这可能会导致低地址部分出现很多小的空闲分区，而每次分配查找时，都要经过这些分区，因此也增加了查找的开销。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。





首次适应算法每次都要从头查找，每次都需要检索低地址的小分区。但是这[illegible]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）综合来看，四种算法中，首次适应算法的效果反而更好

内存分配方式——动态分区分配算法

算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	空闲分区以地址递增次序排列	综合看性能最好。算法开销小，回收分区后一般不需要对空闲分区队列重新排序	
最佳适应	优先使用更小的分区，以保留更多大分区	空闲分区以容量递增次序排列	会有更多的大分区被保留下来，更能满足大进程需求	会产生很多太小的、难以利用的碎片；算法开销大，回收分区后可能需要对空闲分区队列重新排序
最坏适应	优先使用更大的分区，以防止产生太小的不可用的碎片	空闲分区以容量递减次序排列	可以减少难以利用的小碎片	大分区容易被用完，不利于大进程；算法开销大（原因同上）
临近适应	由首次适应演变而来，每次从上次查找结束位置开始查找	空闲分区以地址递增次序排列（可排列成循环链表）	不用每次都从低地址的小分区开始检索。算法开销小（原因同首次适应算法）	会使高地址的大分区也被用完

分页与分段

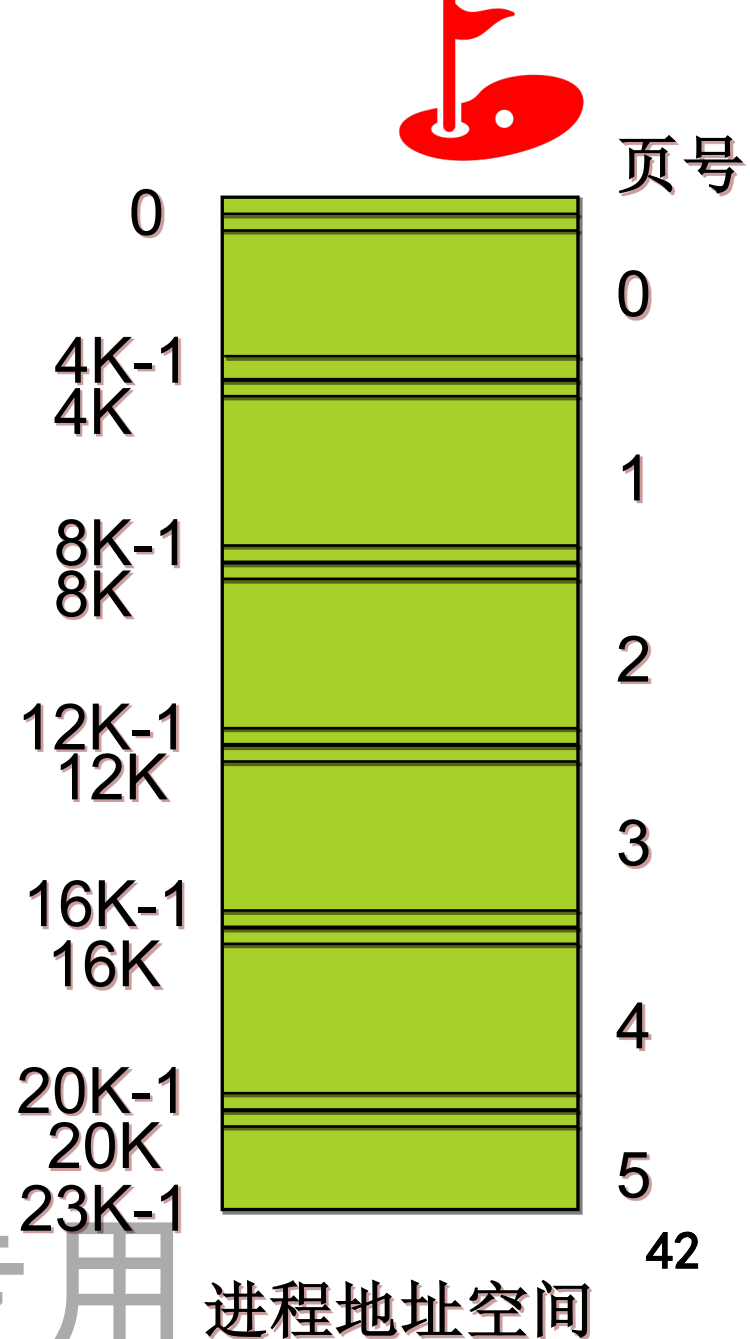
基本分页存储管理的基本概念（页面的概念）

将进程逻辑空间划分为若干等长的区域，称为页（或页面）；并对每个页面顺序编号，称为页号。

页面大小：

页面的大小是2的幂：2^nB

通常为512B～8KB



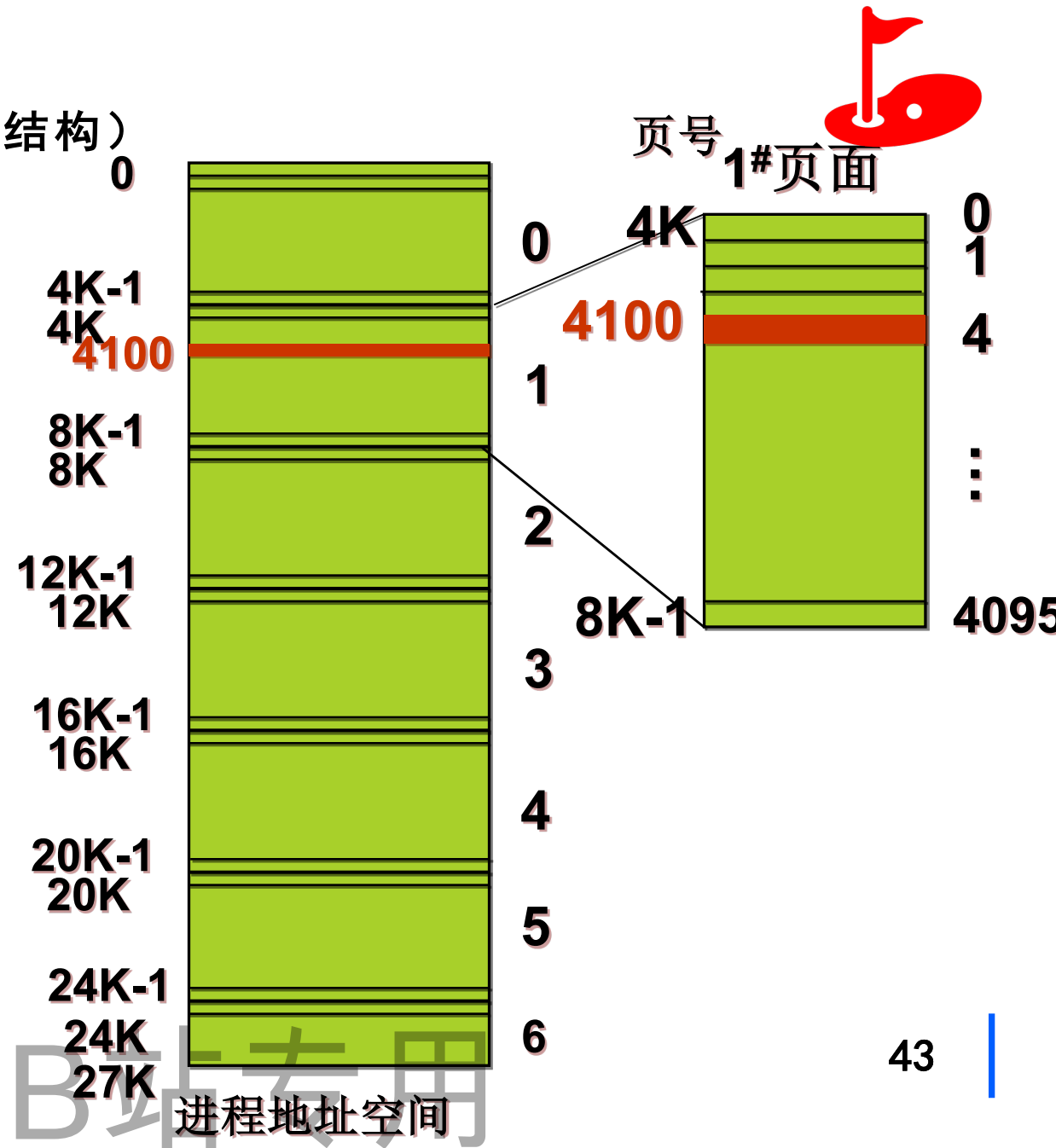
基本分页存储管理的基本概念（逻辑地址结构）

4100#单元

1#页面	4#单元

页号 P	页内 位移d

一维线性地址



基本分页存储管理的基本概念（逻辑地址结构）

- 页号P和页内地址d的计算公式
  - P＝INT [A/L]　　INT：整除函数
  - d＝[A] % L　　%：取余
  - A：逻辑地址空间中的地址，L：页面大小

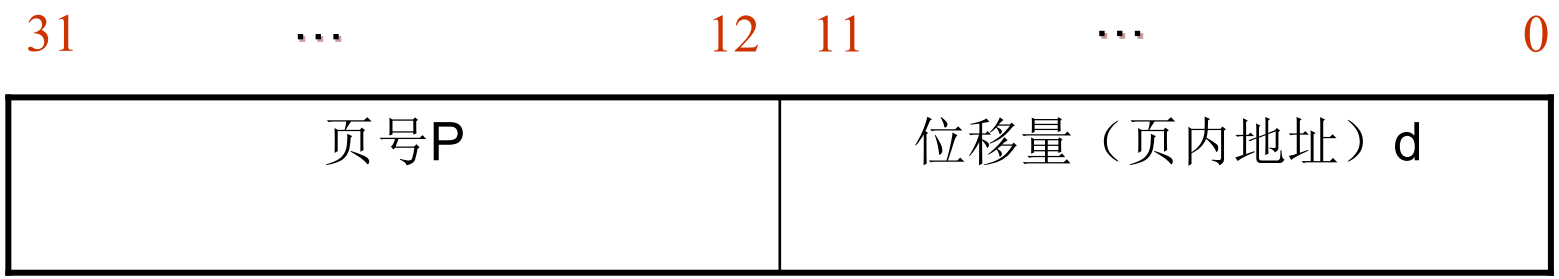
例如：某系统的页面大小为1KB，地址A=2170B，则求得：

P=2，d=122

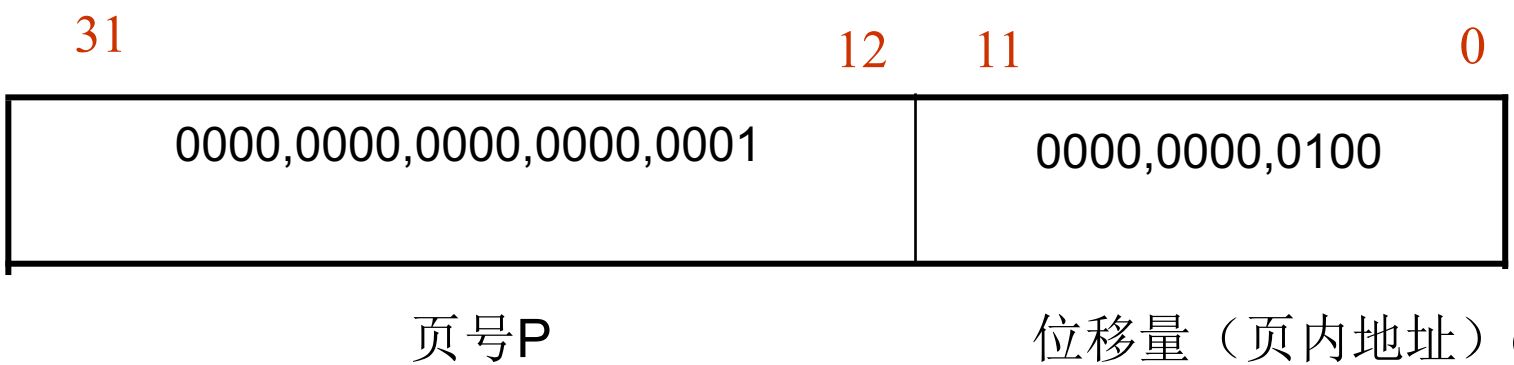
基本分页存储管理的基本概念（逻辑地址结构）

页面大小是2的幂：$=2^nB$

当页面大小为4KB时：$= 2^{12}B$



例如，页面大小为4KB时，逻辑地址4100可表示为：



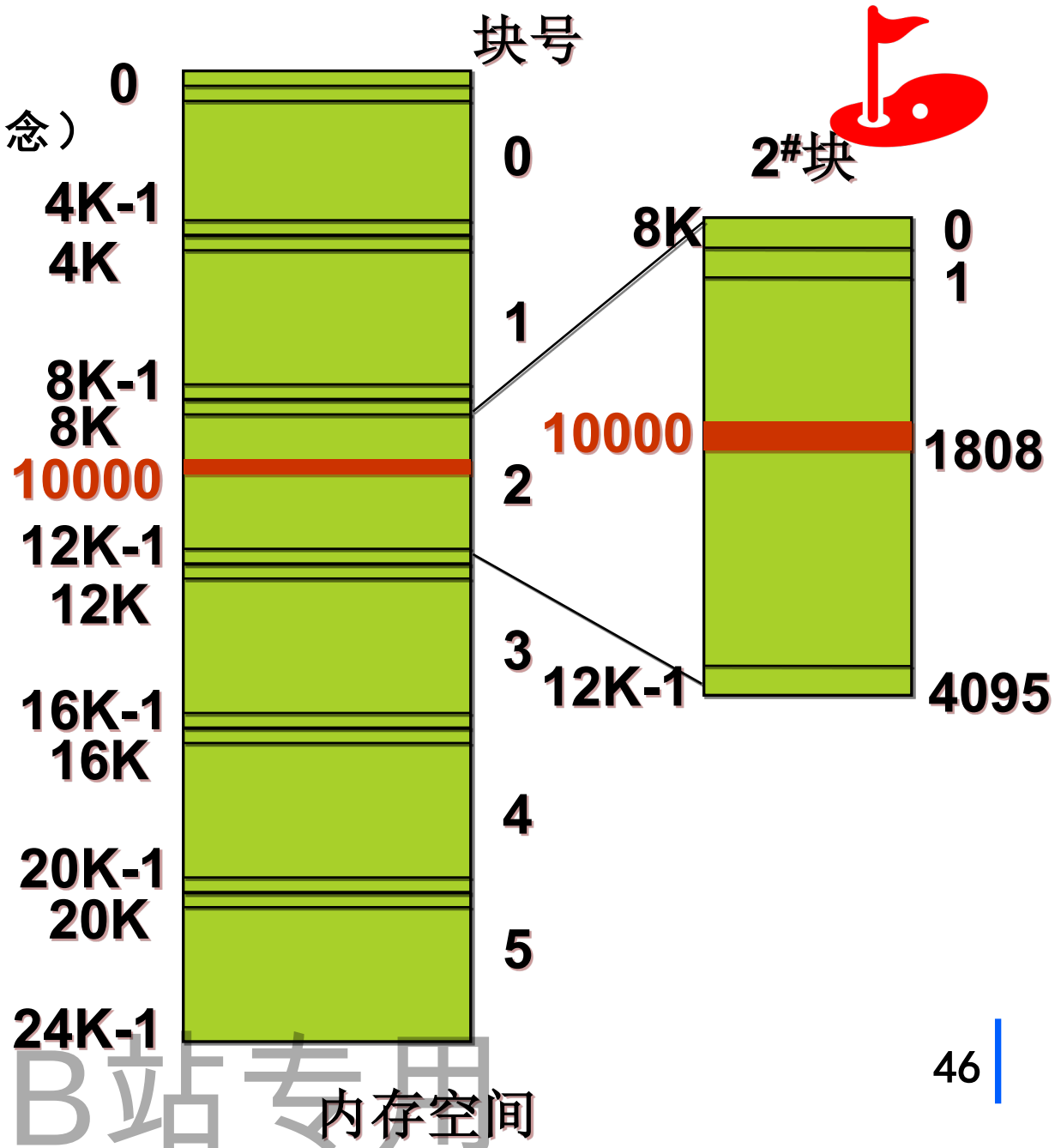
基本分页存储管理的基本概念（物理块概念）

物理块的概念：

将内存空间划分为与页面等长的若干区，称为物理块或页框。

物理地址结构：

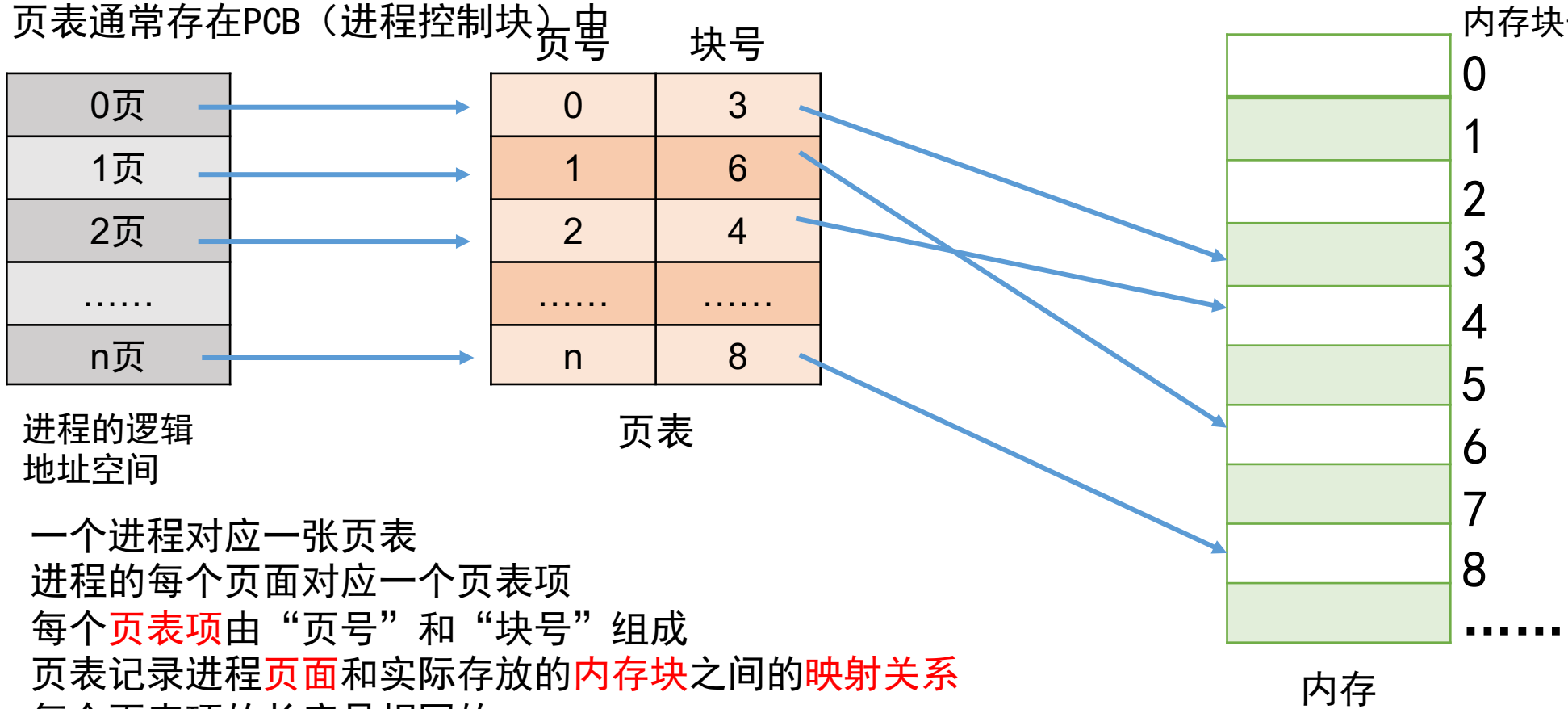
块号	块内位移



基本分页存储管理的基本概念（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。

注：页表通常存在PCB（进程控制块）中



1. 一个进程对应一张页表
2. 进程的每个页面对应一个页表项
3. 每个页表项由“页号”和“块号”组成
4. 页表记录进程页面和实际存放的内存块之间的映射关系
5. 每个页表项的长度是相同的

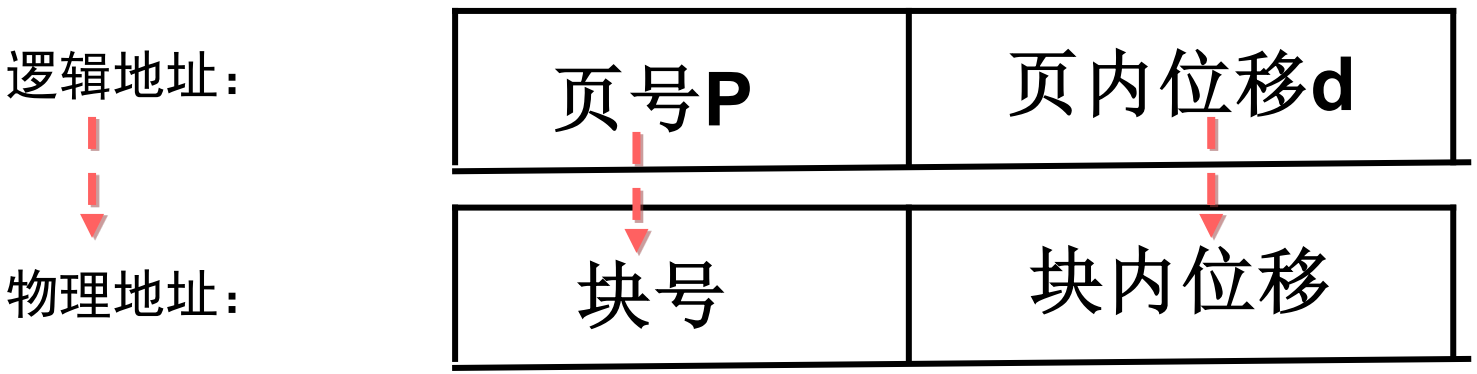
基本分页存储管理的基本概念（地址变换机构）

地址变换机构的任务：实现地址映射，即实现从逻辑地址到物理地址的变换过程。

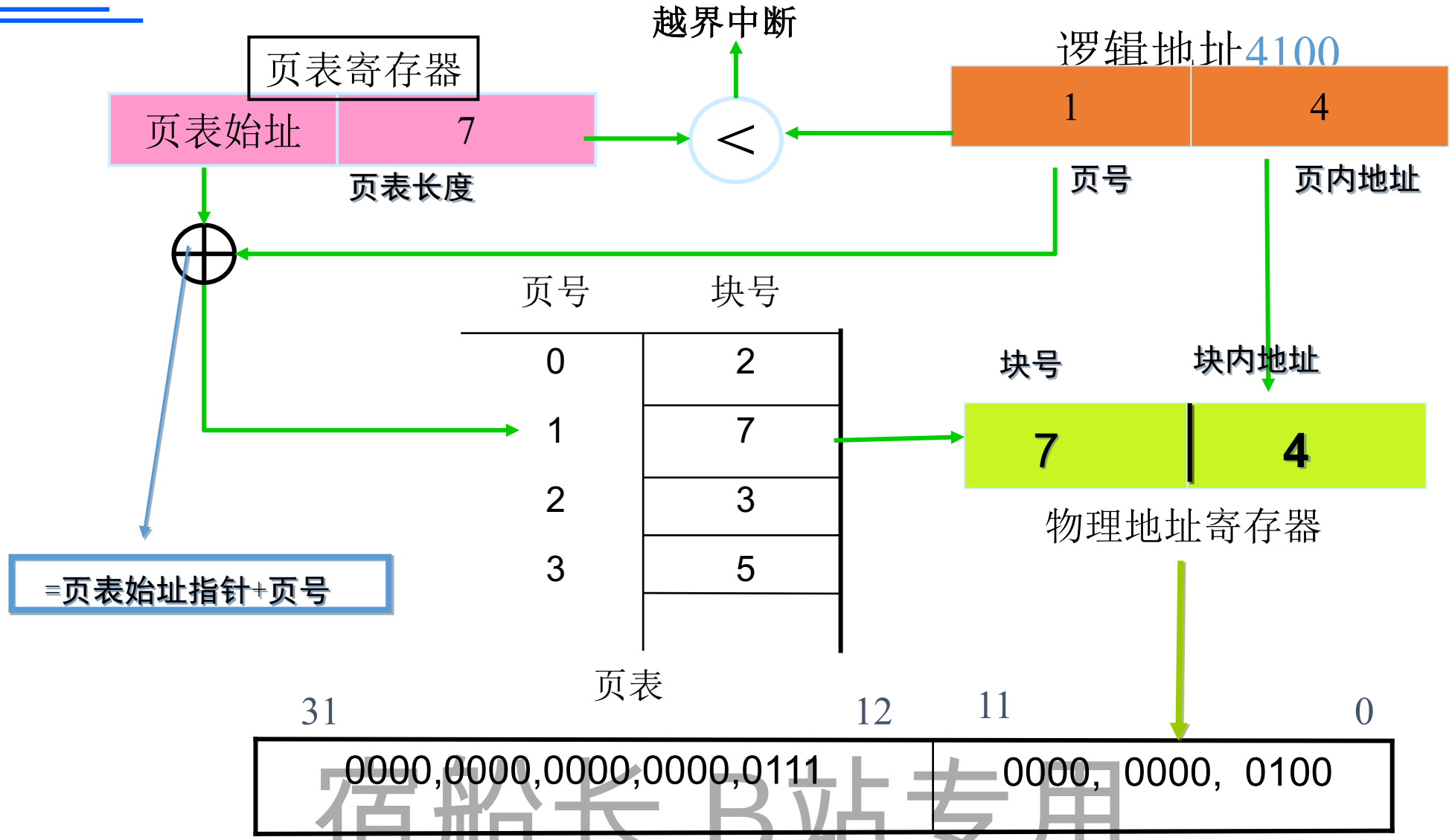
页表：存放在内存系统区的一个连续空间中；

PCB：存有进程页表在内存的首地址和页表长度；

页表寄存器PTR：存放当前进程页表在内存的首地址和页表长度；

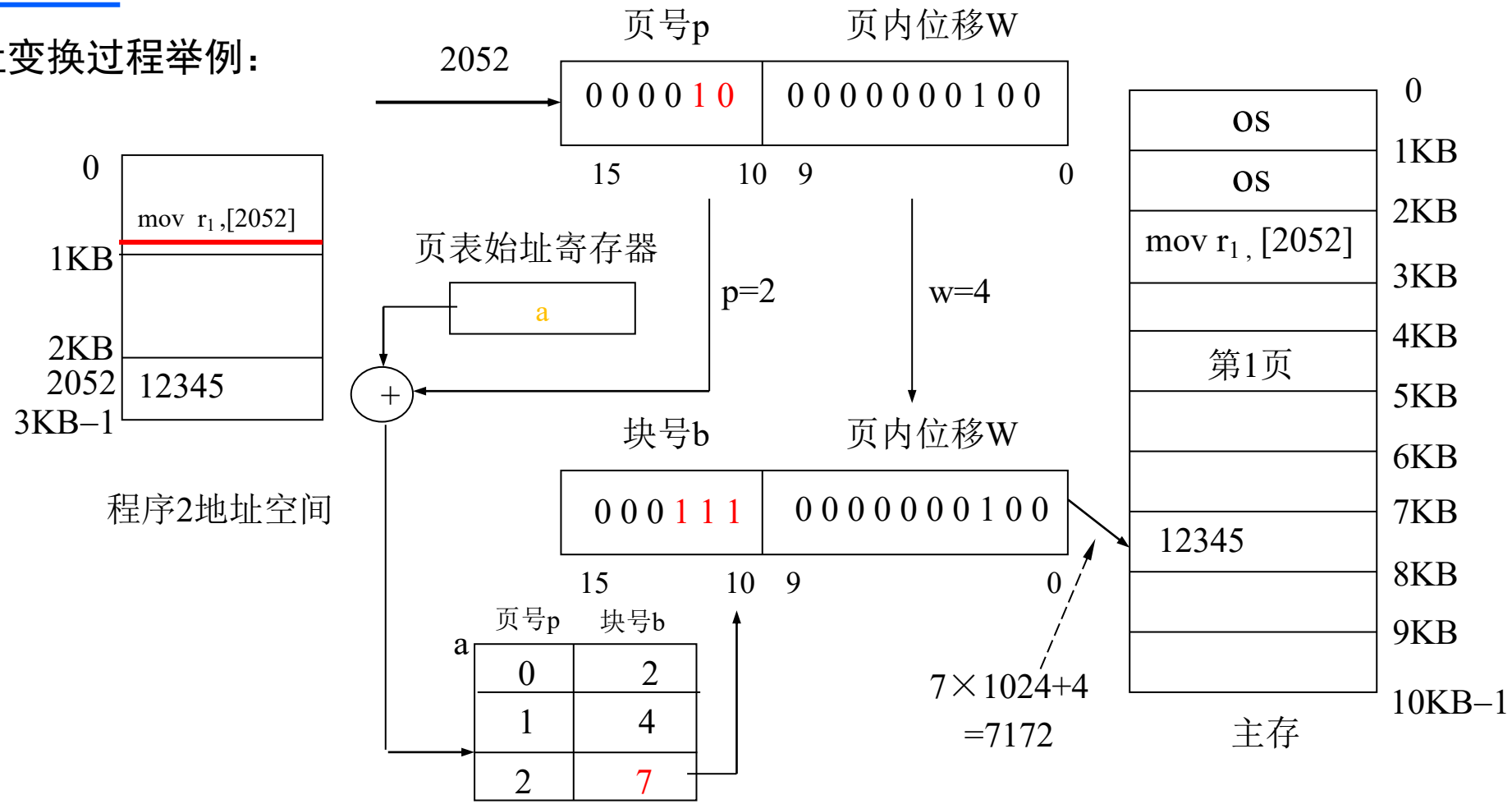


基本分页存储管理的基本概念（地址变换机构）



基本分页存储管理的基本概念（地址变换机构）

页式地址变换过程举例：



页式系统的地址变换过程

基本分页存储管理的基本概念（具有快表的地址变换机构）

快表（联想存储器，按内容查找）：

具有并行查询能力的高速缓冲寄存器

快表大小：

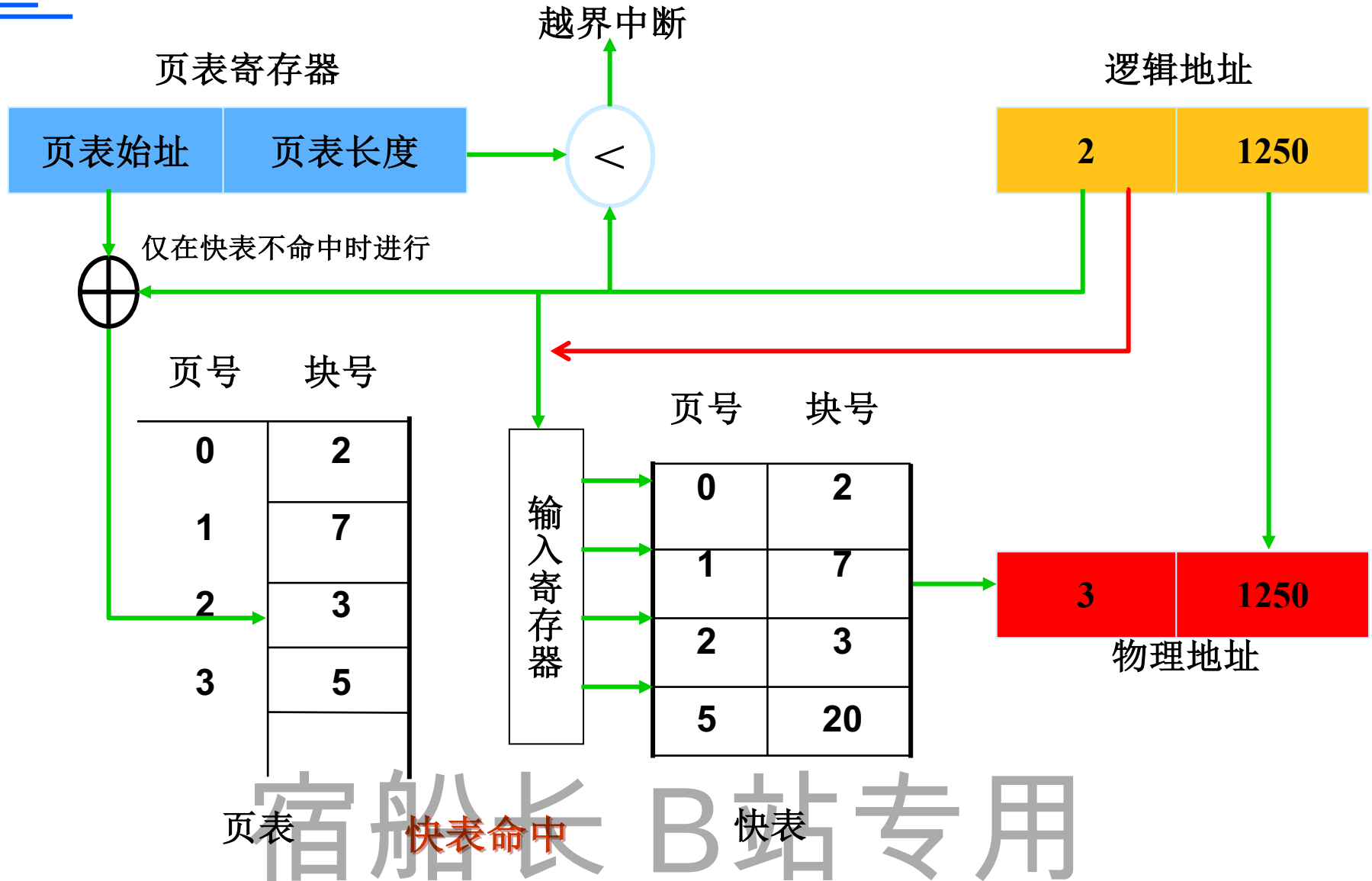
几K到几百K ，只含有部分页表项（16~512个）

如：intel x86:32项

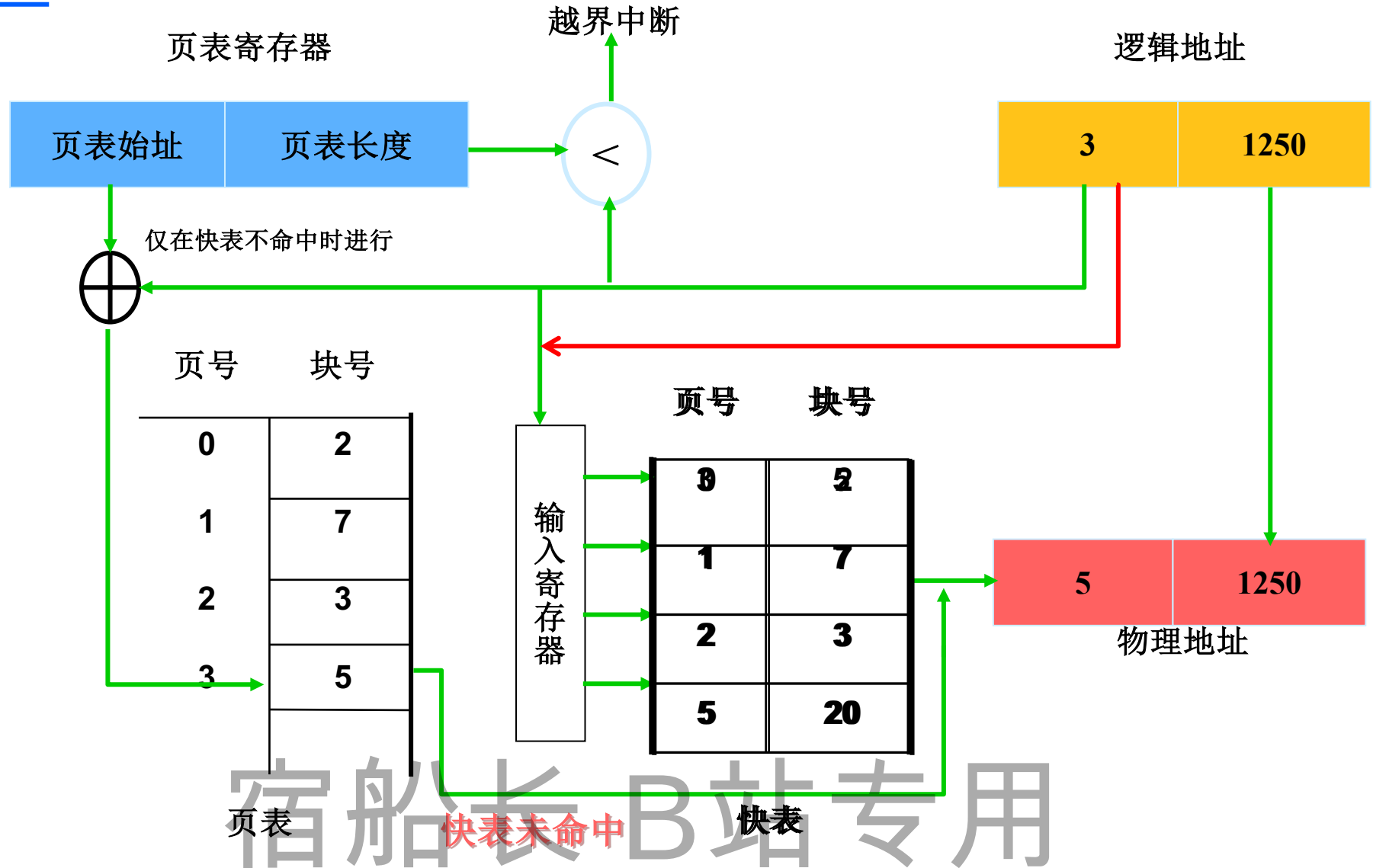
Linux快表表项：

每一个TLB条目包含一页的信息：有效位、虚页号、修改位、保护位和页所在的物理页号

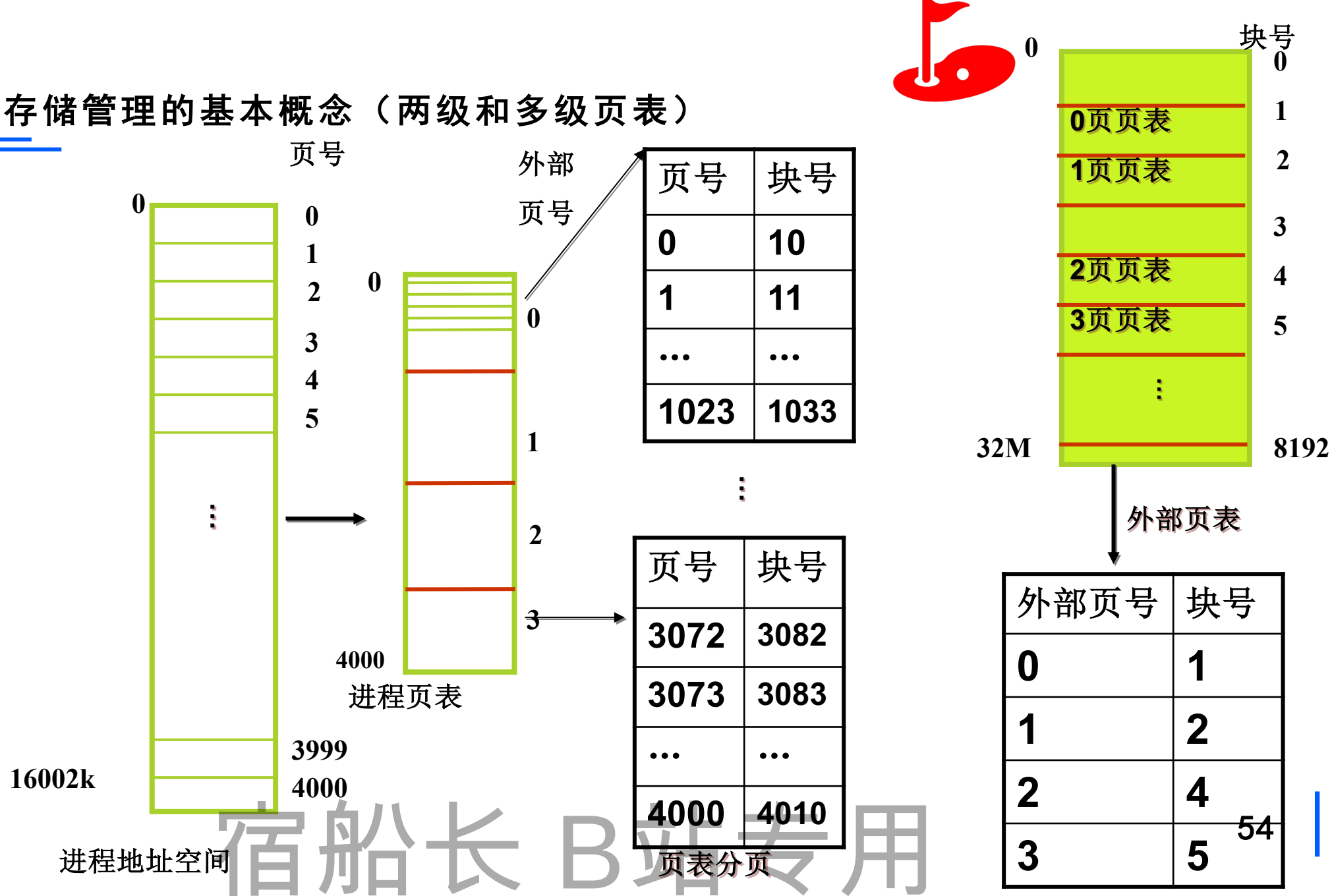
基本分页存储管理的基本概念（具有快表的地址变换机构）



基本分页存储管理的基本概念（具有快表的地址变换机构）



基本分页存储管理的基本概念（两级和多级页表）



基本分页存储管理的基本概念（两级和多级页表）

进程页表：

页号	块号
0	10
1	11
…	…
1023	1033
1024	1034
1025	1035
…	…
2047	2057
2048	2058
2049	2059
…	…
3071	3081
3072	3092
3073	3093
…	…
4000	4010

0#页表分页：1024个页表项

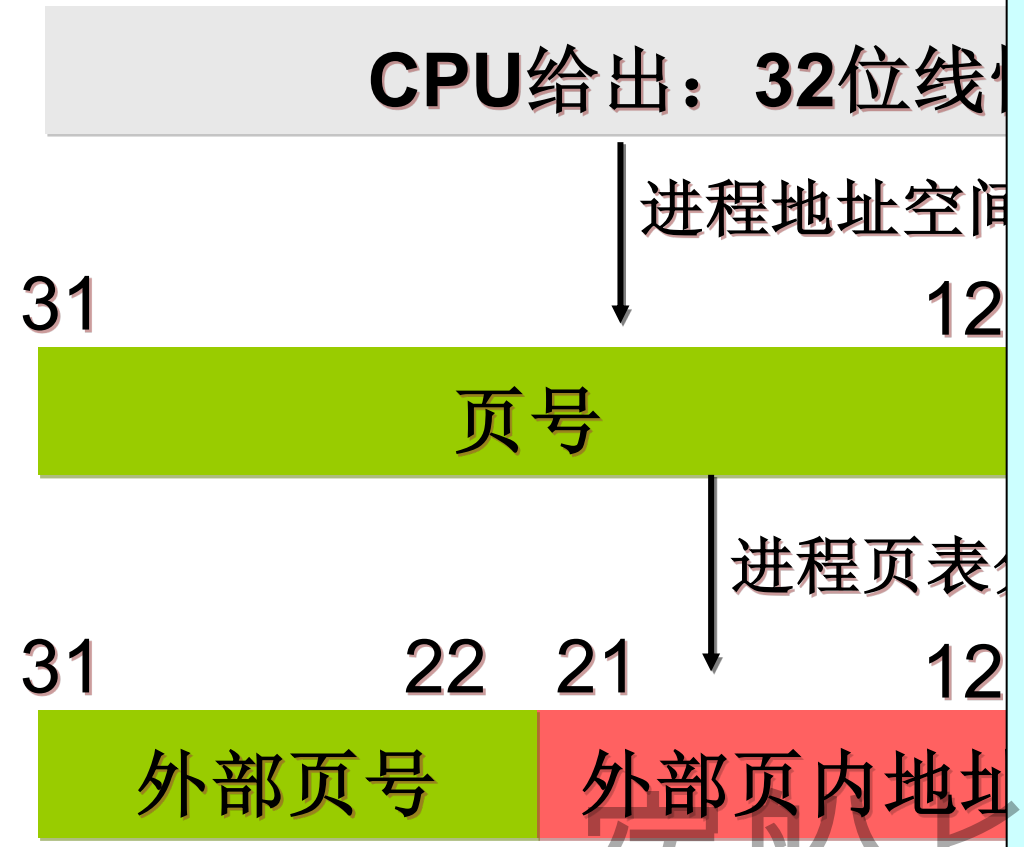
1#页表分页：1024个页表项

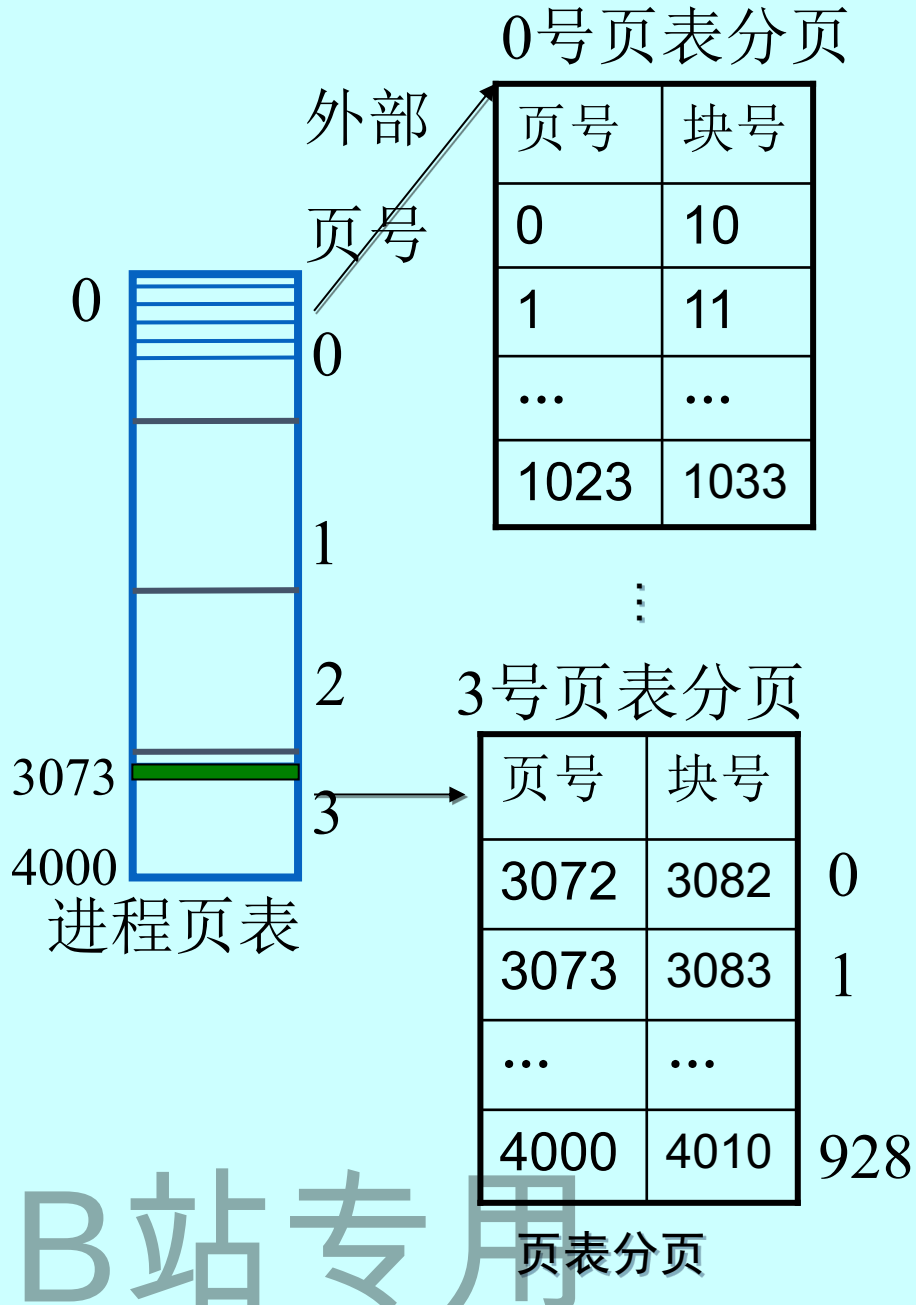
2#页表分页：1024个页表项

3#页表分页：929个页表项

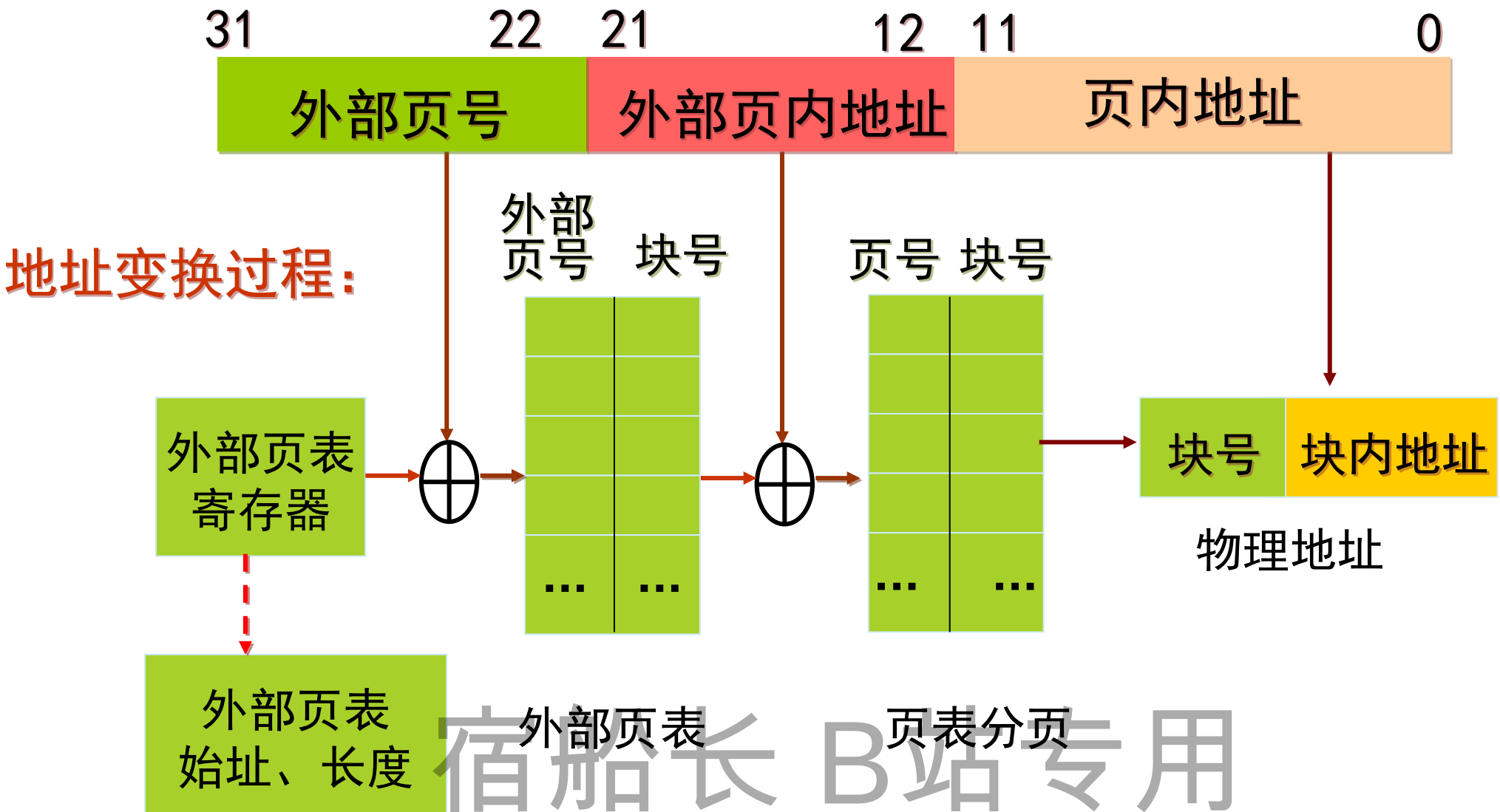
基本分页存储管理的基本概念（两级和

地址结构：

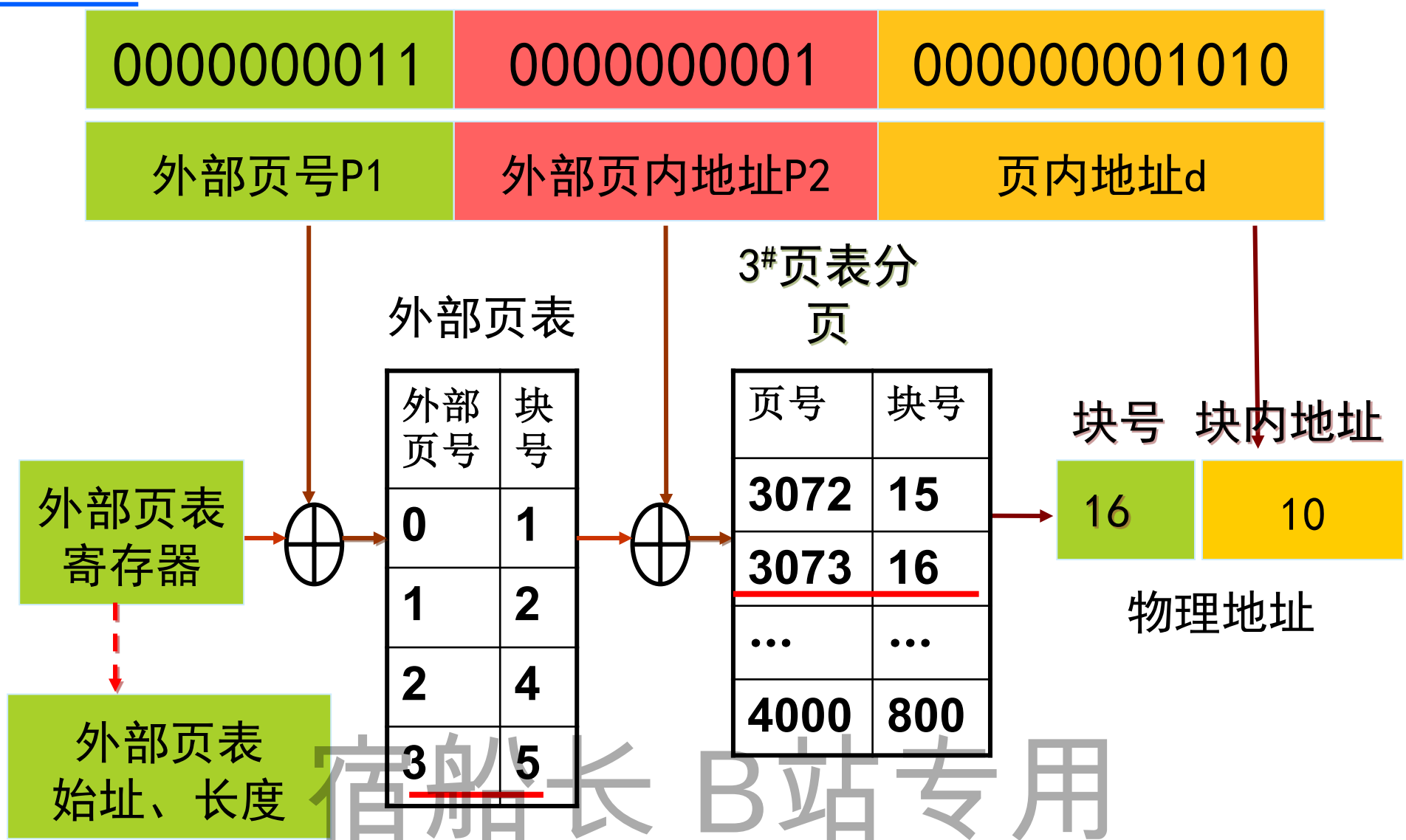




基本分页存储管理的基本概念（两级和多级页表）



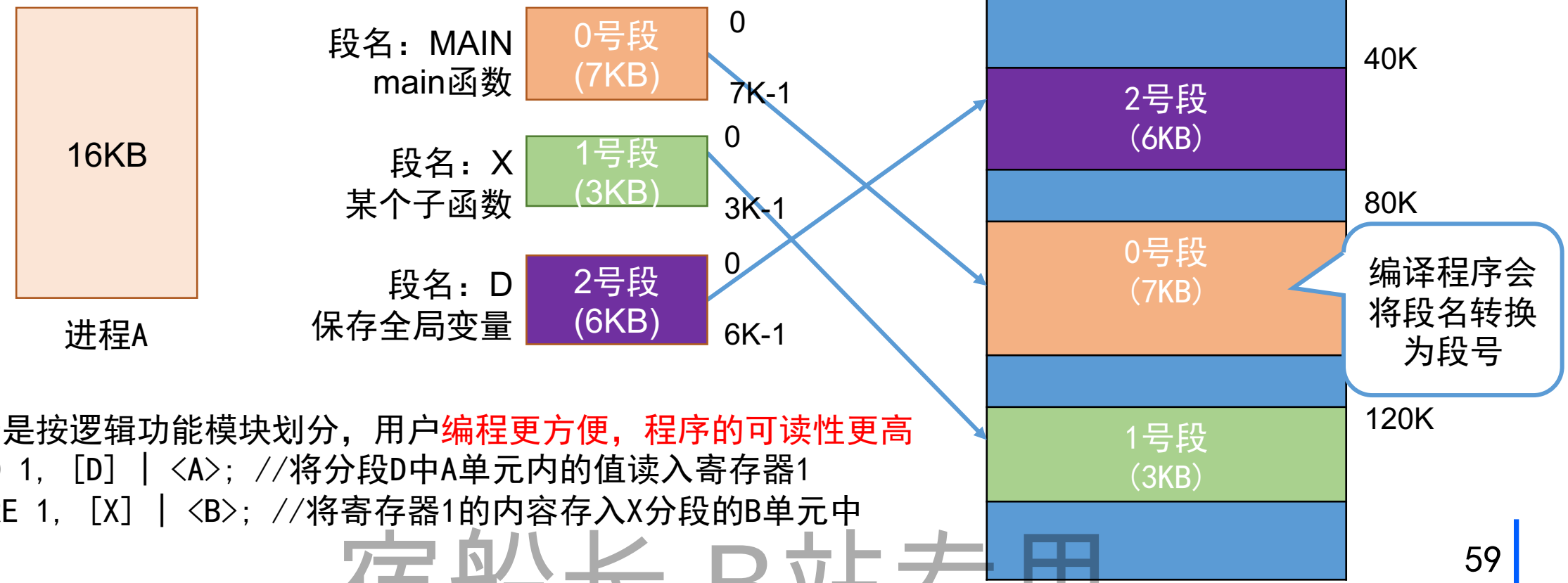
基本分页存储管理的基本概念（两级和多级页表）



基本分段存储管理方式（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。



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
```

基本分段存储管理方式（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。

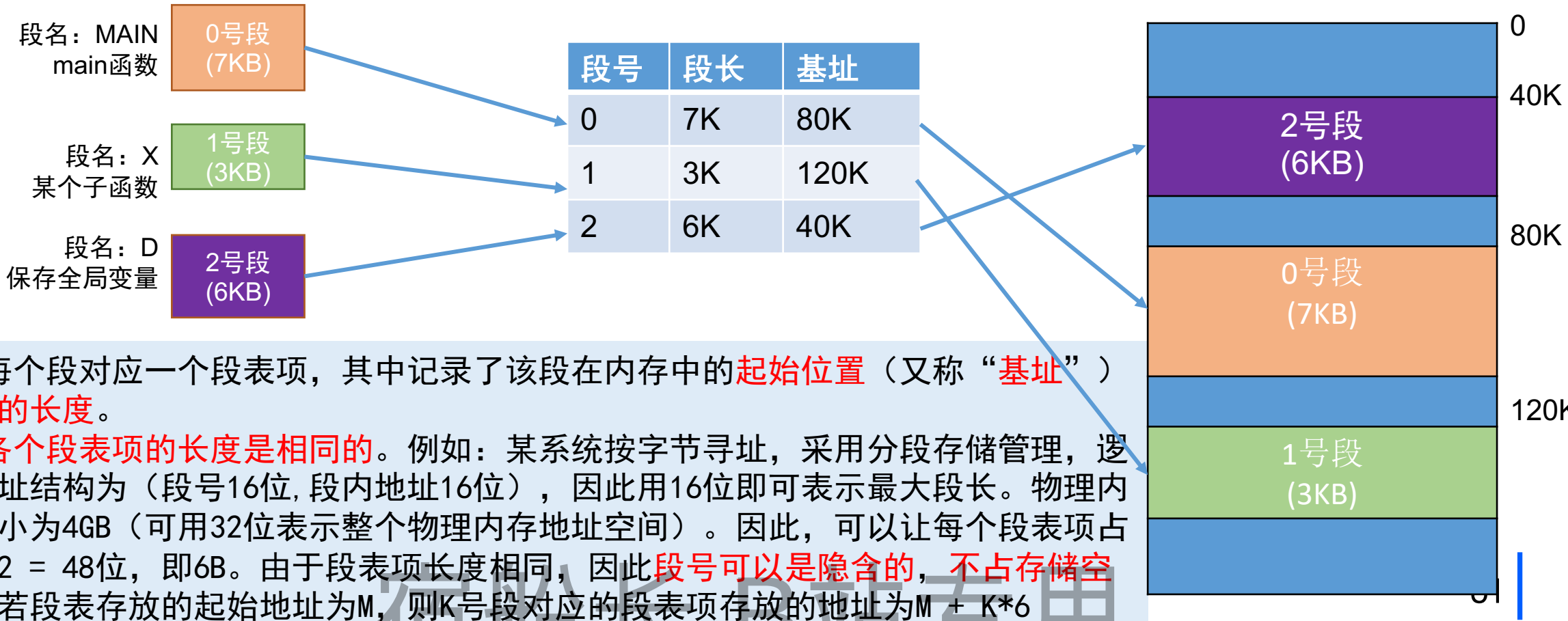
```
LOAD 1, [D] | <A>; //将分段D中A单元内的值读入寄存器1
STORE 1, [X] | <B>; //将寄存器1的内容存入X分段的B单元中
```

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

基本分段存储管理方式（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

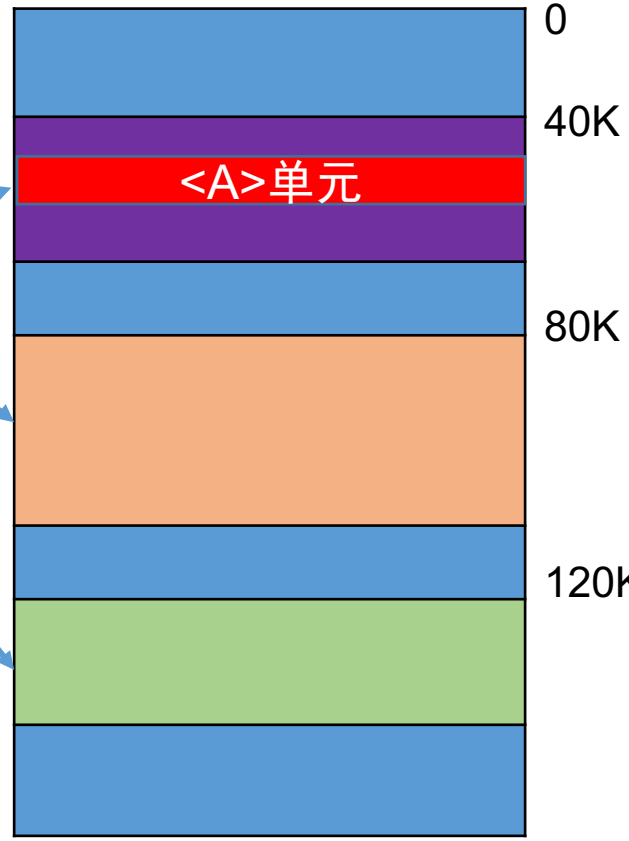
基本分段存储管理方式（地址变换）

段名：MAIN
main函数

段名：X
某个子函数

段名：D
保存全局变量

段号	段长	基址
0	7K	80K
1	3K	120K
2	6K	40K



LOAD 1, [D] | <A>; //将分段D中A单元内的值读入寄存器1

经过编译程序编译后，形成等价的机器指令：
“取出段号为2，段内地址为1024的内存单元中的内容，放到寄存器1中”

CPU执行指令时需要将逻辑地址变换为物理地址

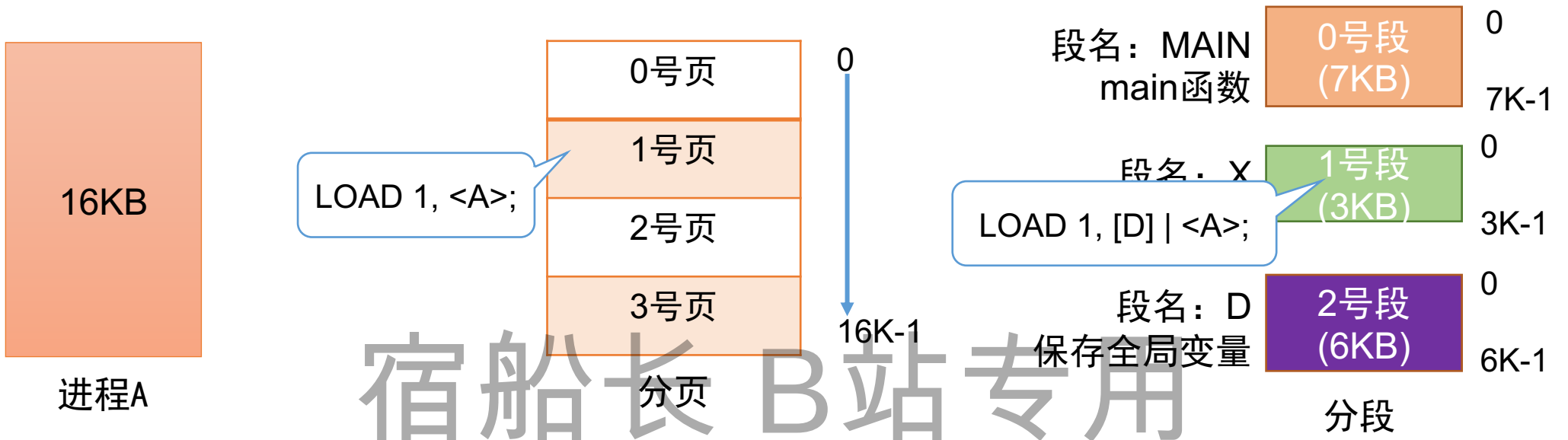
机器指令中的逻辑地址用二进制表示：0000000000000010000000100000000

基本分段存储管理方式（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。



基本分段存储管理方式（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。

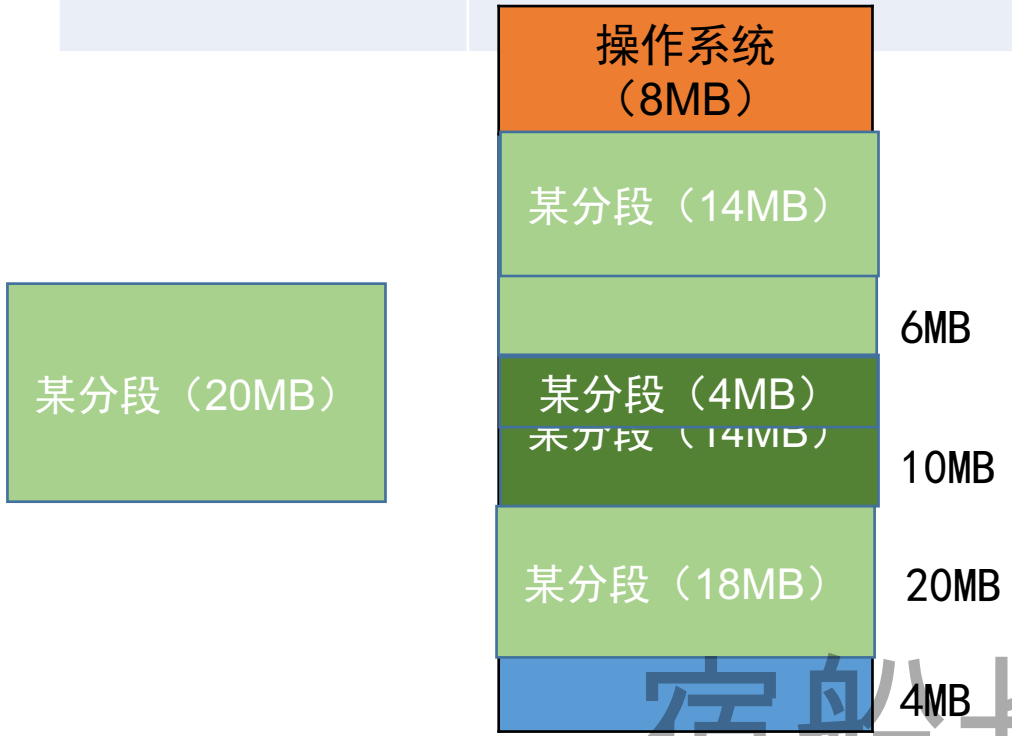
页的大小固定且由系统决定。段的长度却不固定，决定于用户编写的程序。
分页的用户进程地址空间是一维的，程序员只需给出一个记忆符即可表示一个地址。
分段的用户进程地址空间是二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。

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

访问一个逻辑地址需要几次访存？
分页（单级页表）：第一次访存——查内存中的页表，第二次访存——访问目标内存单元。总共两次访存
分段：第一次访存——查内存中的段表，第二次访存——访问目标内存单元。总共两次访存
与分页系统类似，分段系统中也可以引入快表机构，将近期访问过的段表项放到快表中，这样可以少一次访问，加快地址变换速度。

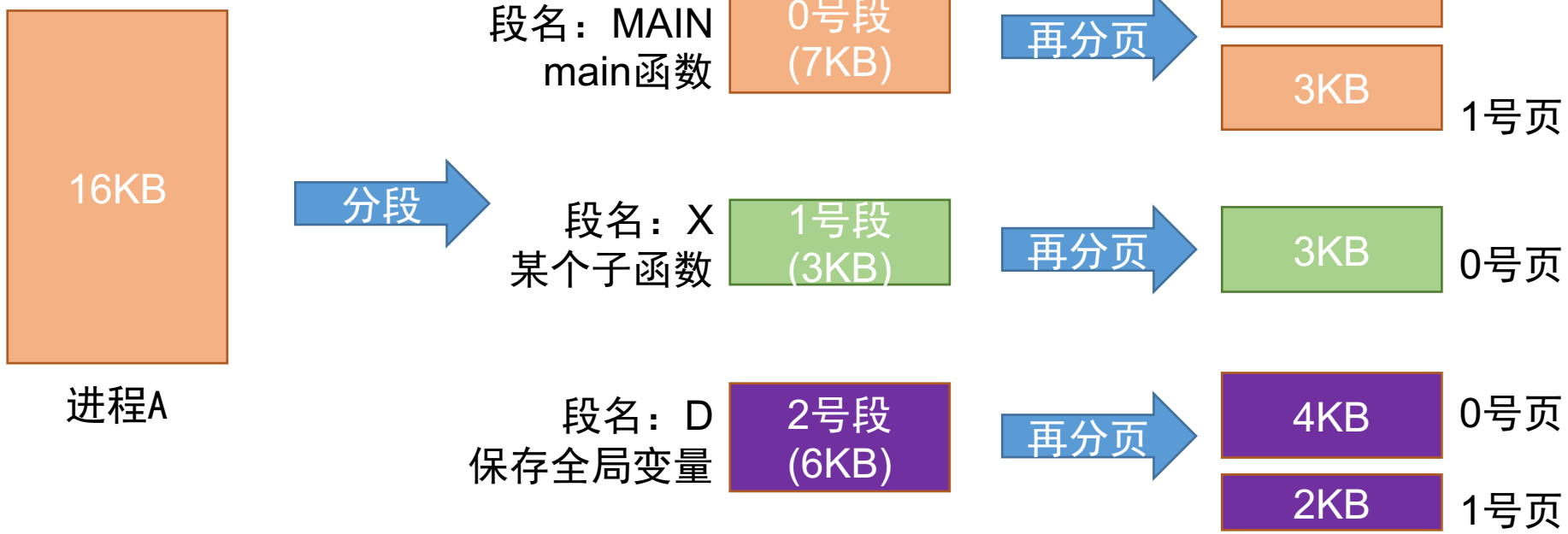
段页式管理方式（分页、分段的优缺点分析）

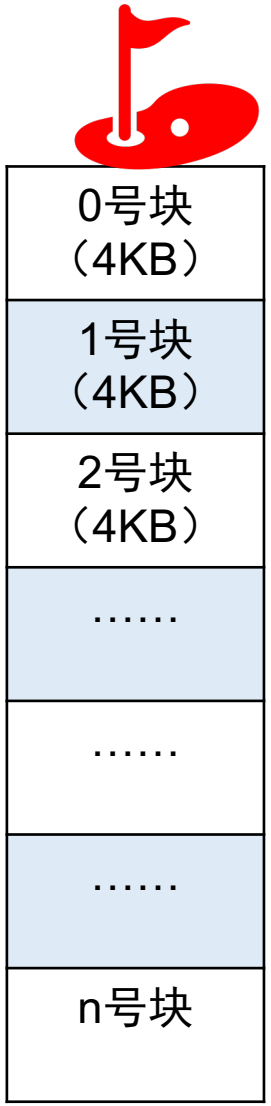
	优点	缺点
分页管理	内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理会产生外部碎片



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

段页式管理方式（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
进程前将各页面分别装入各内存块中

段页式管理方式（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：

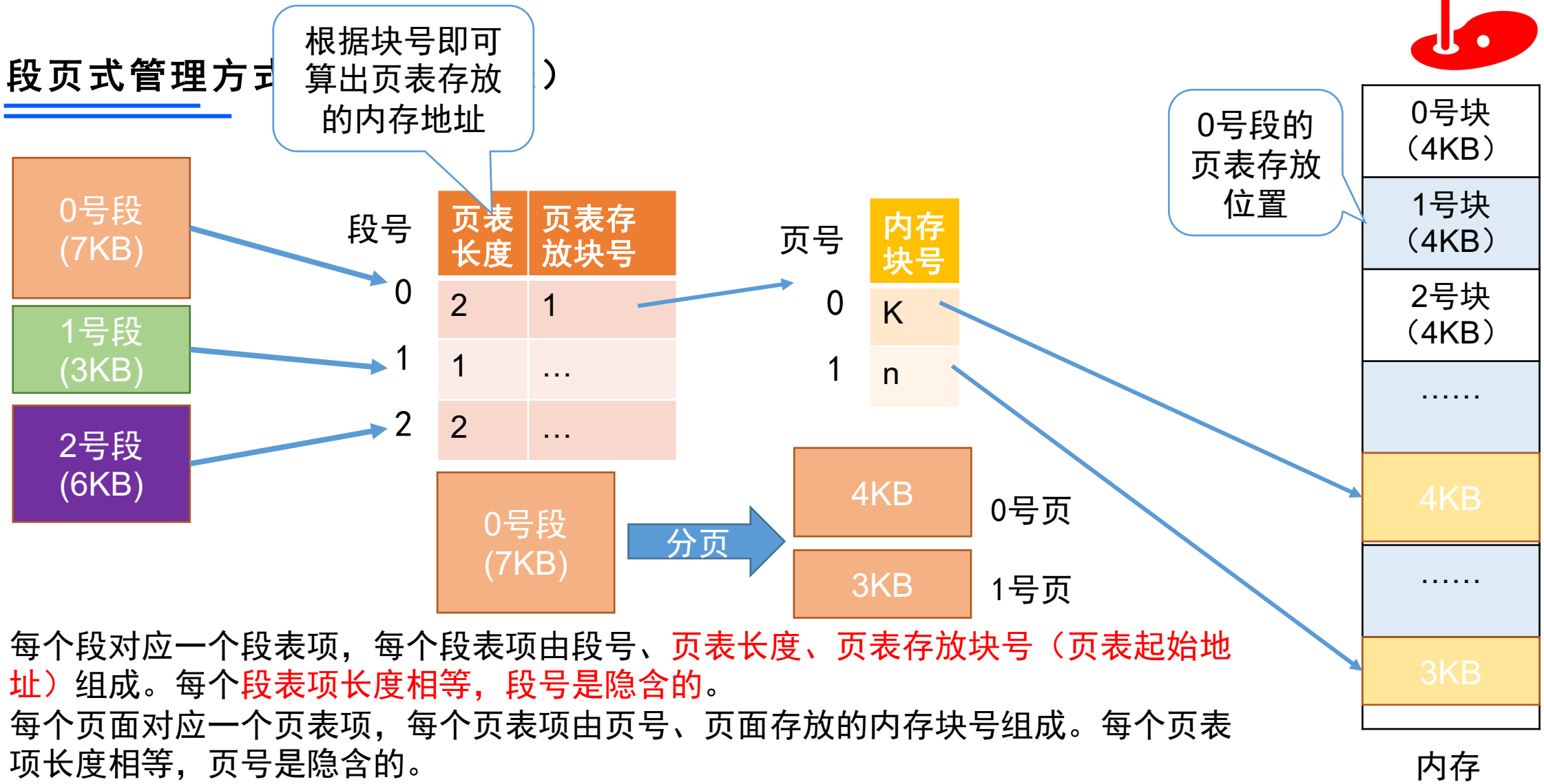
31 …… 16	15 …… 12	11 …… 0
段号	页号	页内偏移量

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 =

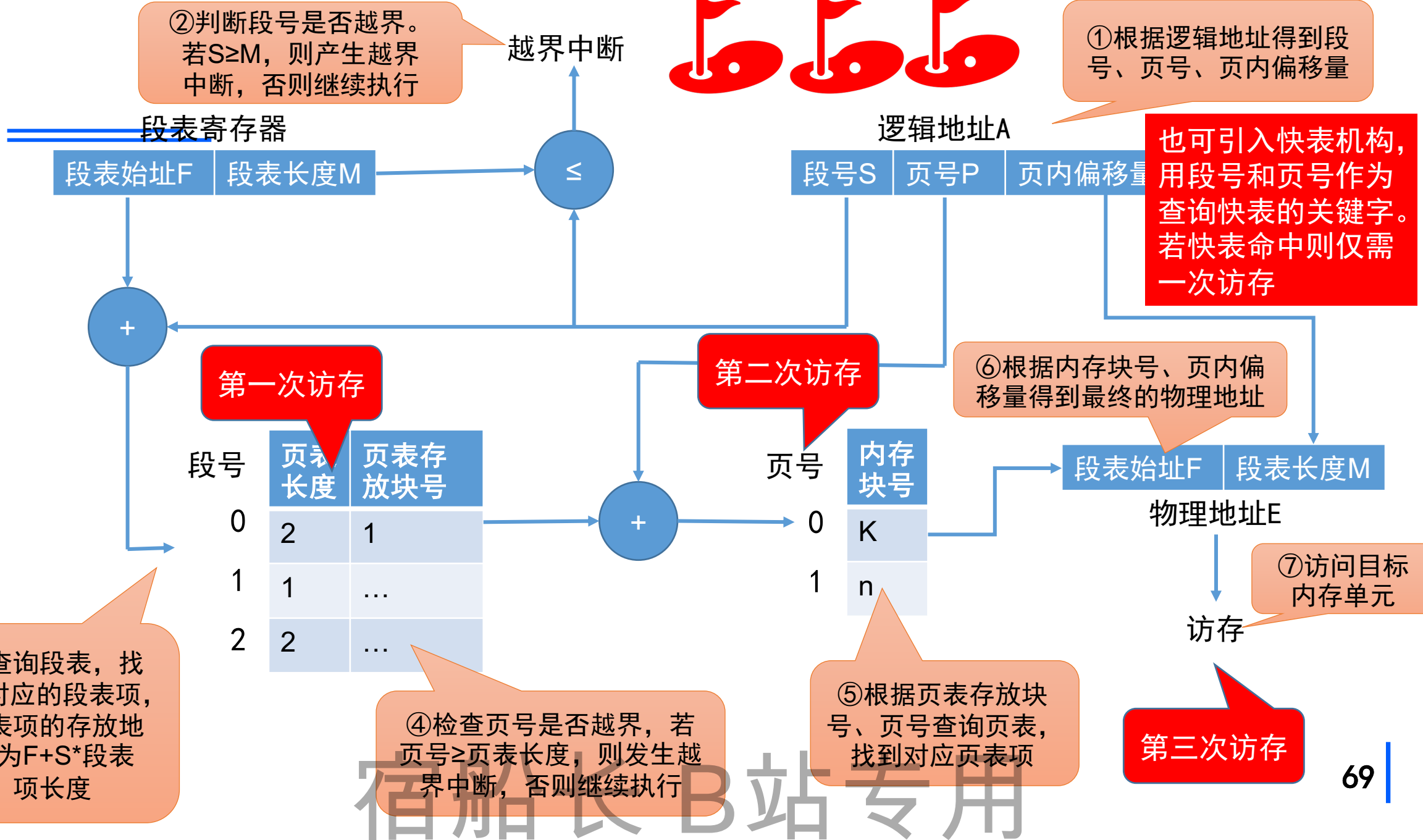
“分段”对用户是可见的，程序员编程时需要显式地给出段号、段内地址。而将各段“分页”对用户是不可见的。系统会根据段内地址自动划分页号和页内偏移量。
因此段页式管理的地址结构是二维的

段页式管理方式



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。


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
越界中断
①根据逻辑地址得到段号、页号、页内偏移量
段表寄存器
逻辑地址A
段表始址F
段表长度M
≤
段号S
页号P
页内偏移量
也可引入快表机构，用段号和页号作为查询快表的关键字。若快表命中则仅需一次访存
+
第一次访存
第二次访存
⑥根据内存块号、页内偏移量得到最终的物理地址
段号
页表长度
页表存放块号
0
2
1
1
1
…
2
2
…
+
页号
内存块号
0
K
1
n
段表始址F
段表长度M
物理地址E
访存
⑦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
④检查页号是否越界，若页号≥页表长度，则发生越界中断，否则继续执行
⑤根据页表存放块号、页号查询页表，找到对应页表项
第三次访存