- 连续分配是指为一个用户进程分配一个连续的内存空间。可进一步分为：
  - 单一连续分配
  - 固定分区分配
  - 动态分区分配
  - 动态重定位分区分配

4.3.1 单一连续分配

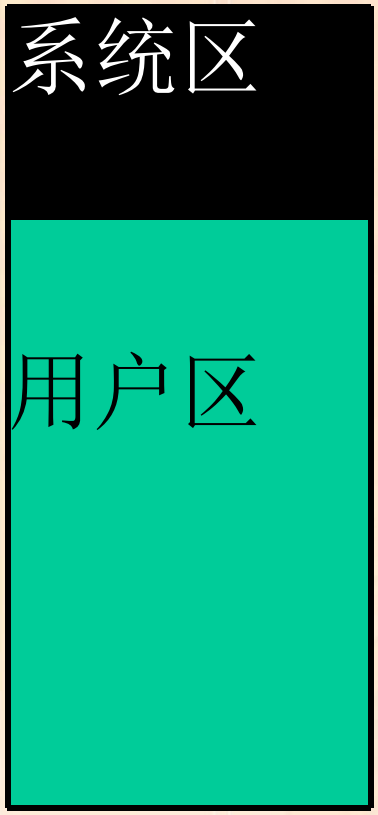
- 内存分为两个区域：系统区，用户区。应用程序装入到用户区，可使用用户区全部空间。未采取存储保护措施。
- 最简单，适用于单用户、单任务的OS。CP/M和MS-DOS
- 优点：
  - 易于管理。
- 缺点：
  - 对要求内存空间少的程序，造成内存浪费；
  - 程序全部装入，很少使用的程序部分也占用内存

分区式存储管理

- 为了支持多道程序系统和分时系统，支持多个程序并发执行，引入了分区式存储管理。
- 分区式存储管理是把内存分为一些大小相等或不等的分区，操作系统占用其中一个分区，其余的分区由应用程序使用，每个应用程序占用一个或几个分区。
- 内碎片和外碎片：
  - 前者是占用分区之内未被利用的空间
  - 后者是占用分区之间难以利用的空闲分区（通常是小空闲分区）。

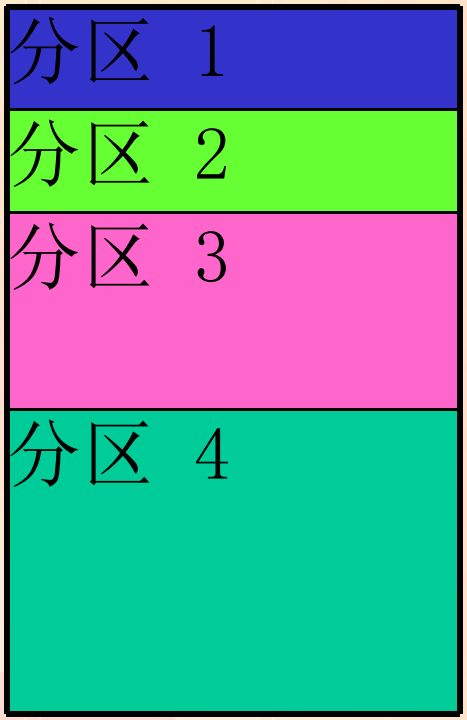
4.3.2 固定分区分配(fixed partitioning)

- 最简单的一种运行多道程序的存储管理方式
- 把内存划分为若干个固定大小的连续分区，每个分区只装入一个作业。
- 划分分区的方法
  - 分区大小相等：只适合于多个相同进程的并发执行（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。



单一连续分配

固定分区分配（分区相等）



固定分区分配（分区不等）

4.3.2 固定分区分配(fixed partitioning)

- 内存分配
  - OS将分区按大小进行排队，并建立一张分区使用表或位示图。
  - 可以和覆盖、交换技术配合使用运行较大的用户进程。

分区号	大小(k)	起址(k)	状态
1	12	20	已分配
2	32	32	未分配
3	64	64	已分配
4	128	128	已分配

- 优点：
  - 易于实现，开销小。
- 缺点：
  - 内碎片造成浪费，分区总数固定，限制了并发执行的程序数目。

4.3.3 动态分区分配(dynamic partitioning)

动态分配

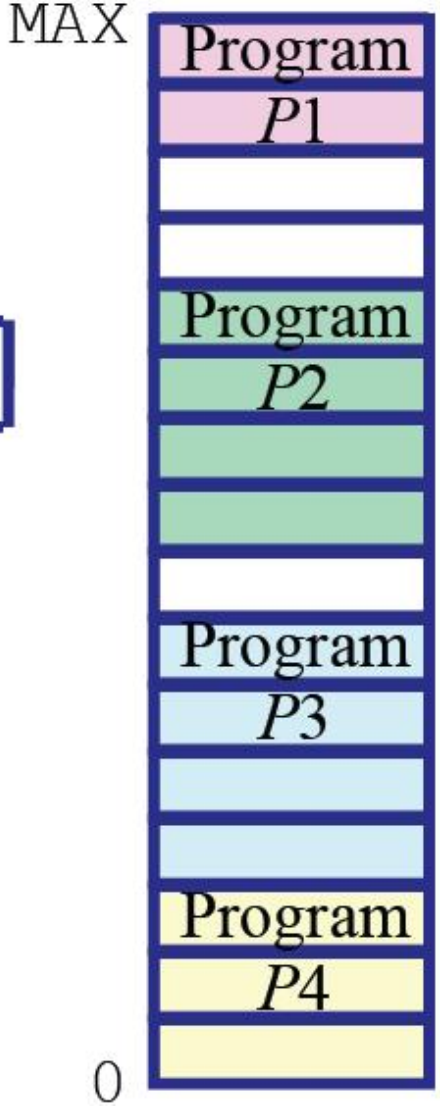
- 简单的内存管理方式:
  - 当程序被允许进入系统的时候，被分配内存块
  - 分配的内存块在地址上是连续的

块=内存块=分区

操作系统持续追踪...

- 程序运行中的所有已分配内存块
- 目前空闲的内存块 Empty-blocks ("holes")

分配策略
First-fit 首次适配
Best-fit 最佳适配
Worst-fit 最差适配



4.3.3 动态分区分配(dynamic partitioning)

- 动态分区分配是指OS根据进程的实际需要为各进程分配连续的物理内存。
- 分区分配中的数据结构
  - 为了管理内存空闲分区建立了空闲分区表或空闲分区链表。
  - 表中各表项一般包括每个分区的起始地址、大小及状态(是否已分配)。
  - 分区表中，表项数目随着内存的分配和释放而动态改变，可以规定最大表项数目。
  - 分区表可以划分为两个表格：空闲分区表和占用分区表。从而减小每个表格长度。空闲分区表中按不同分配算法对表项排序。

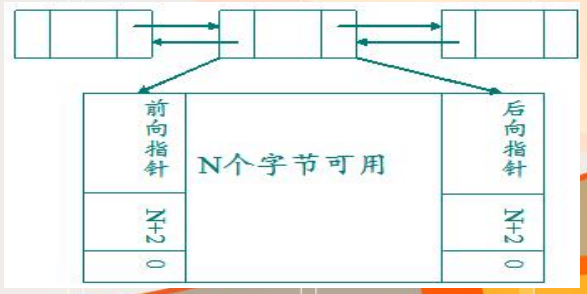
1） 分区分配中的数据结构

（1）空闲分区表示

序号	大小	起址	状态
1	48k	116k	可用
2	252k	260k	可用

（a）空闲分区表

- 空闲分区表：记录每个空闲分区的情况。每个空闲分区占一个表目。
- 空闲分区链：在每个分区的起始部分，设置一些用于控制分区分配的信息，以及用于链接各分区所用的前向指针；在分区尾部则设置一后向指针,将所有的空闲分区链接成一个双向链。



（b）空闲分区链

（2）已占分区说明表

结构：作业号；起始地址；大小

作业号	大小	起址
1	32k	20k
2	64k	52k
4	96K	164K

（C）已占分区表

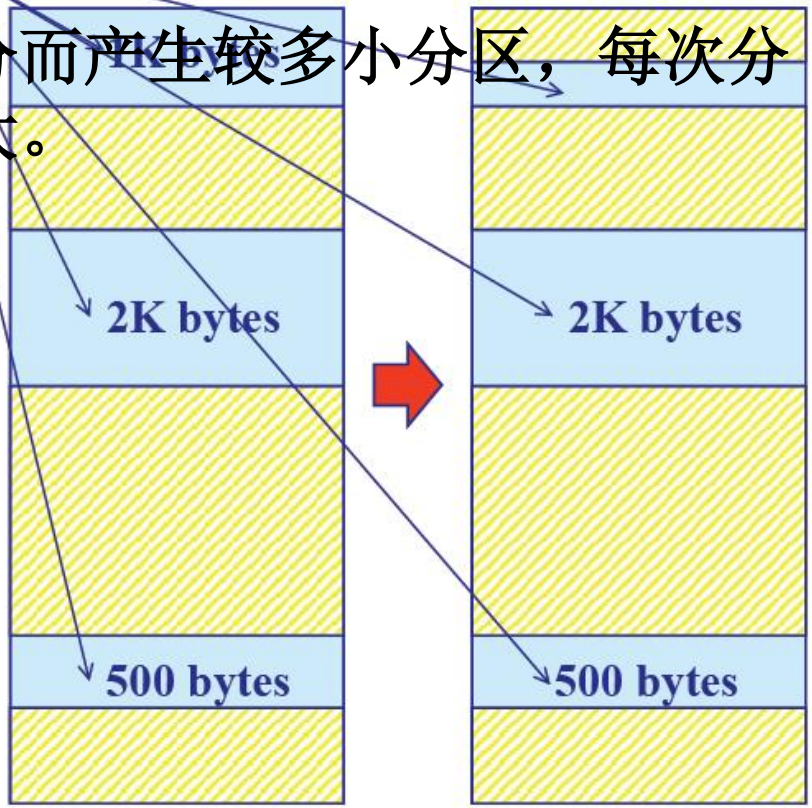
4.3.3 动态分区分配(dynamic partitioning)

- 分区分配算法：
  - 某个新作业装入内存，需寻找一个空闲分区，其大小需大于或等于进程的要求。
  - 若是大于要求，则将该分区分割成两个分区，其中一个分区为要求的大小并标记为“占用”，而另一个分区为余下部分并标记为“空闲”。

4.3.3 动态分区分配(dynamic partitioning)

（1）首次适应算法(first-fit)

- 按分区的先后次序，从头查找，找到符合要求的第一个分区。
- 该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。
- 但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。



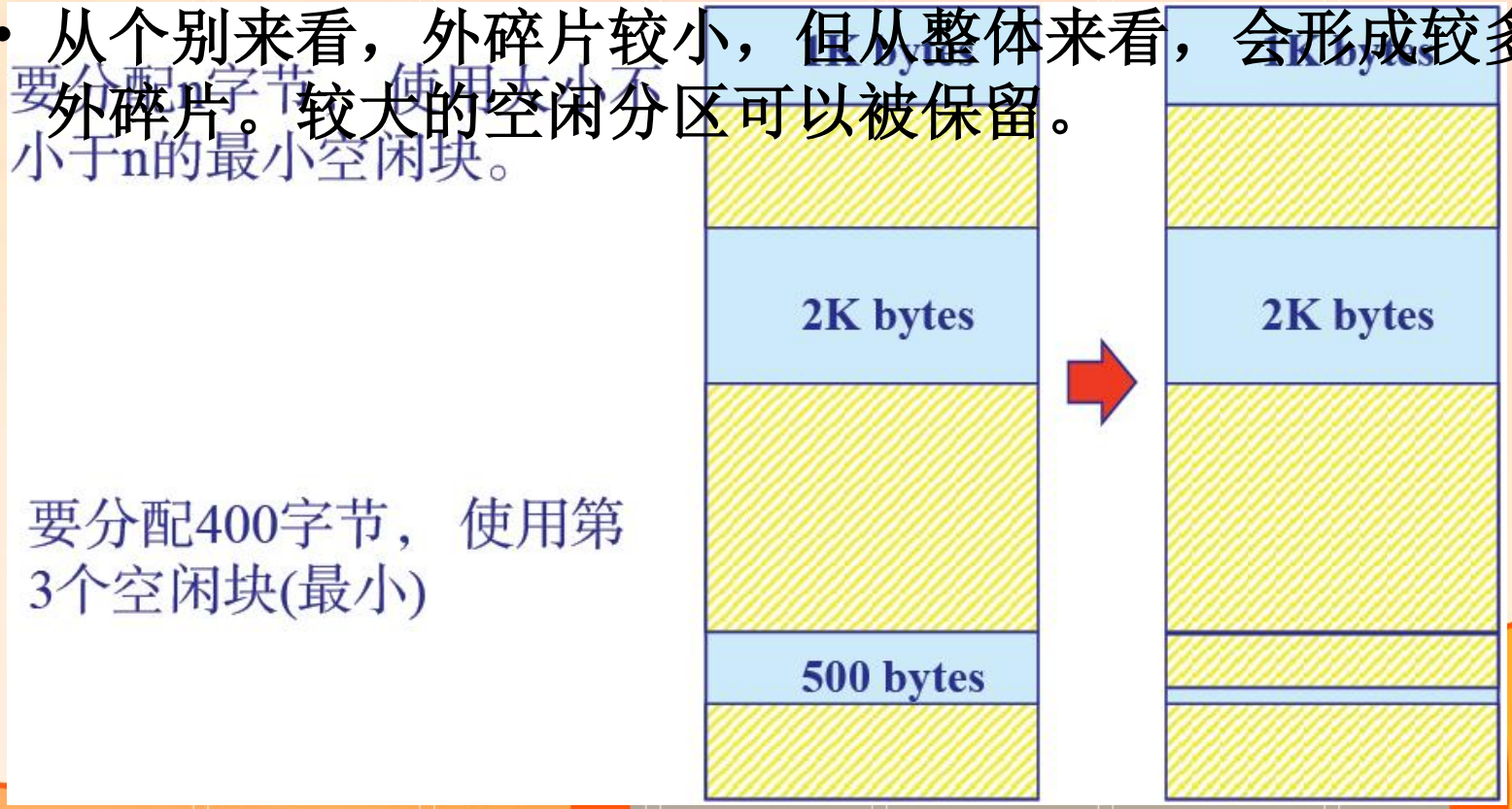
要分配400字节，使用第一个空闲块。

（2）循环首次适应算法（下次适应法next-fit）

- 按分区的先后次序，从上次分配的分区的下一个位置开始查找（到最后一个分区时再回到开头），找到符合要求的第一个分区。
- 实现算法，要设置起始查询指针。
- 该算法的分配和释放的时间性能较好，使空闲分区分布得更均匀，但较大的空闲分区不易保留。

（3）最佳适应法(best-fit)

- 找到其大小与要求相差最小的空闲分区。
- 为了加速寻找，该算法要求空闲分区表将空闲分区按容量由小到大排序。
- 从个别来看，外碎片较小，但从整体来看，会形成较多外碎片。较大的空闲分区可以被保留。

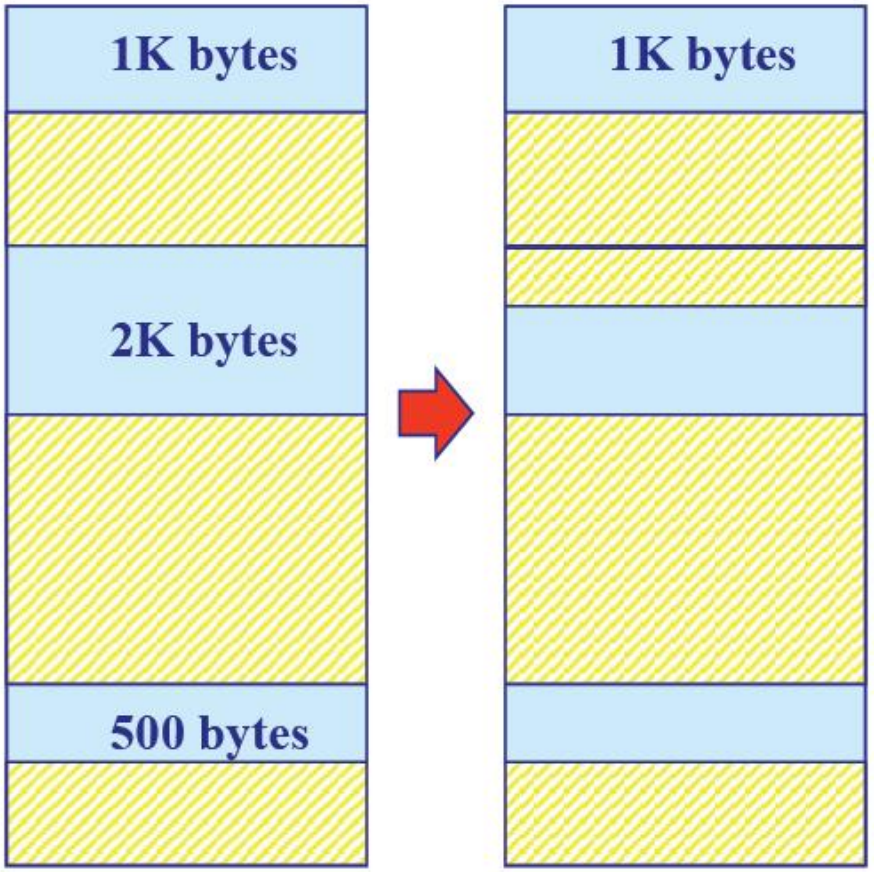


（4）最坏适应法(worst-fit)

- 找到最大的空闲分区。
- 算法要求空闲分区表将空闲分区按容量由大到小排序。
- 基本不留下小空闲分区，但较大的空闲分区不被保留。

要分配n字节，使用尺寸不小于n的最大空闲块

要分配400字节，使用第2个空闲块（最大）



（5）快速适应算法

- 又称分类搜索法。
- 将空闲分区根据容量大小分类，每类分区容量相同。为每类分区设立一个空闲分区链表。系统中设立一张管理索引表，每个表项记录的是每类空闲分区链表的表头。
- 优点：
  - 查找空闲分区效率高。
  - 能保留大分区。
- 缺点：
  - 回收分区时，系统开销大。
  - 空闲分区划分越细，浪费则越严重。

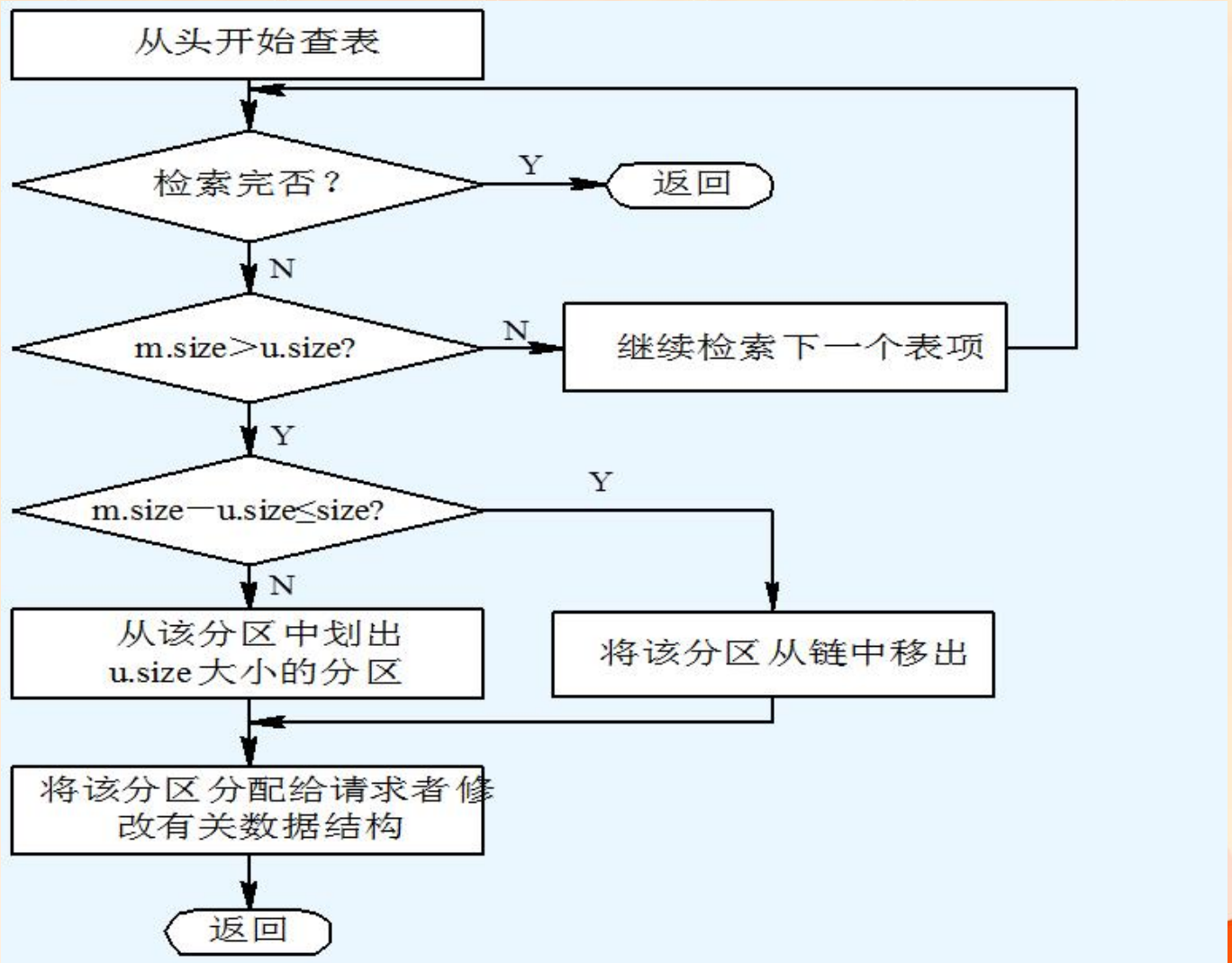
4.3.3 动态分区分配(dynamic partitioning)

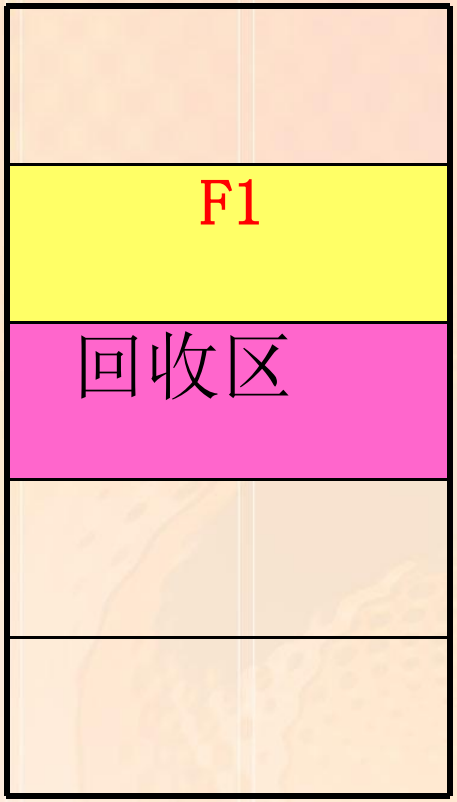
- 分区分配操作
  - 分配内存
    - 利用某种分配算法，从空闲分区表（链）中找到所需大小的分区
  - 回收内存，有以下四种情况：
    - 与前一个空闲分区相邻
    - 与后一个空闲分区相邻
    - 与前、后空闲分区都相邻
    - 不与任何空闲分区相邻

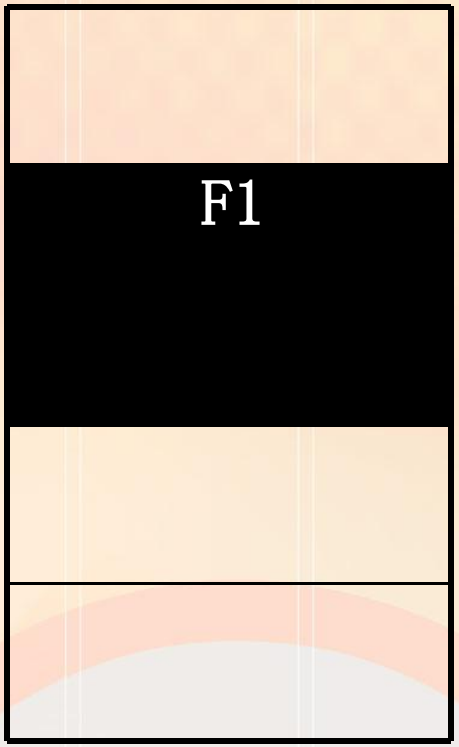
3）分区分配及回收操作

· 分配内存

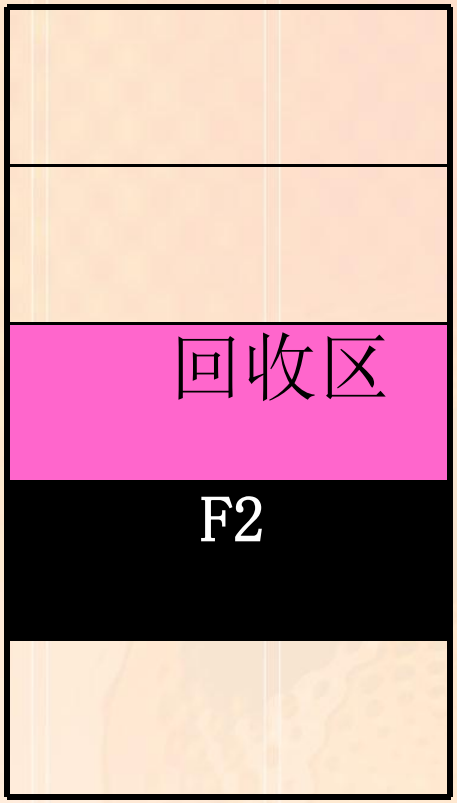
利用某种分配算法，从空闲分区链(表)中找到所需大小的分区。设请求的分区大小为u.size，表中每个空闲分区的大小表示为m.size，若m.size-u.size≤size(规定的不再切割的分区大小)，将整个分区分配给请求者，否则从分区中按请求的大小划出一块内存空间分配出去，余下部分留在空闲链中，将分配区首址返回给调用者。

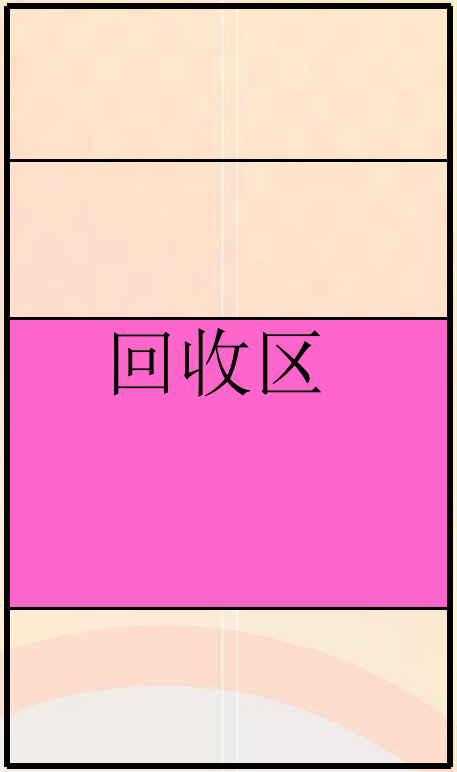

从头开始查表
检索完否？
Y
返回
N
m.size＞u.size?
N
继续检索下一个表项
Y
m.size－u.size≤size?
Y
N
从该分区中划出
u.size大小的分区
将该分区从链中移出
将该分区分配给请求者修
改有关数据结构
返回



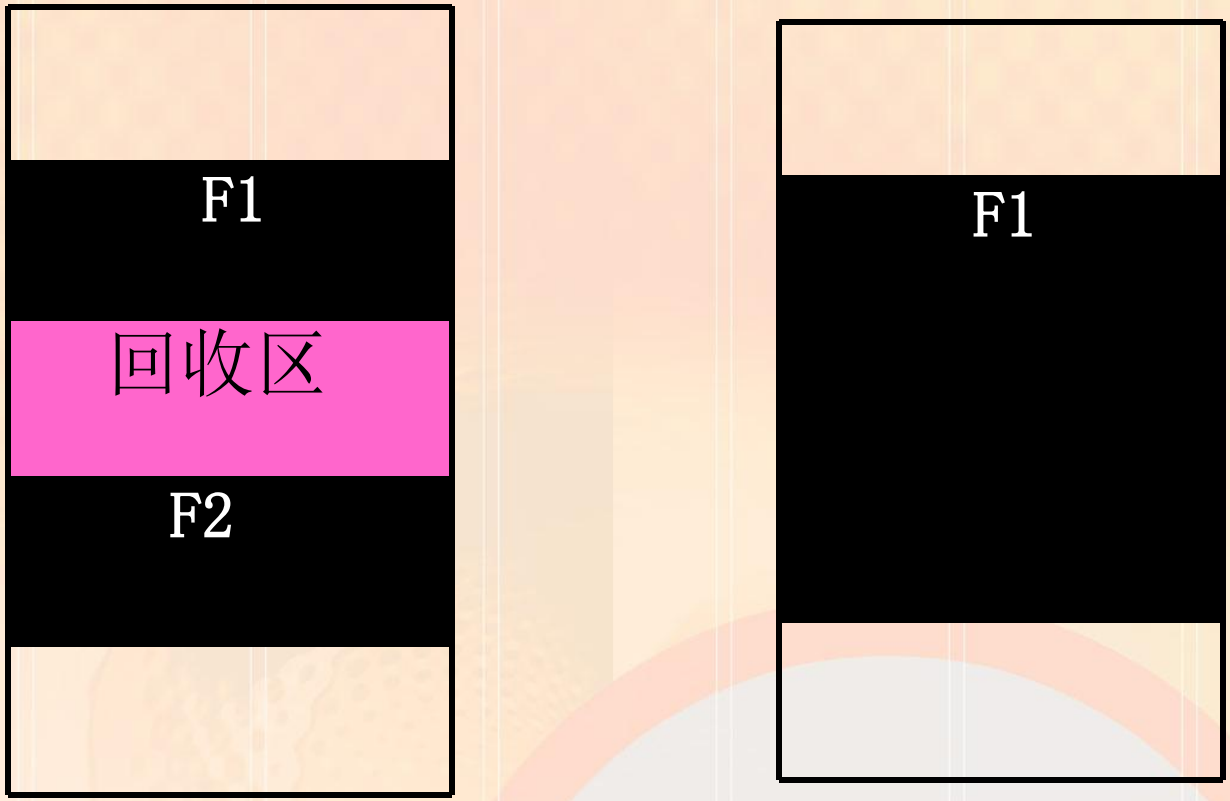


回收区与插入点的前一个空闲区**F1**相邻接





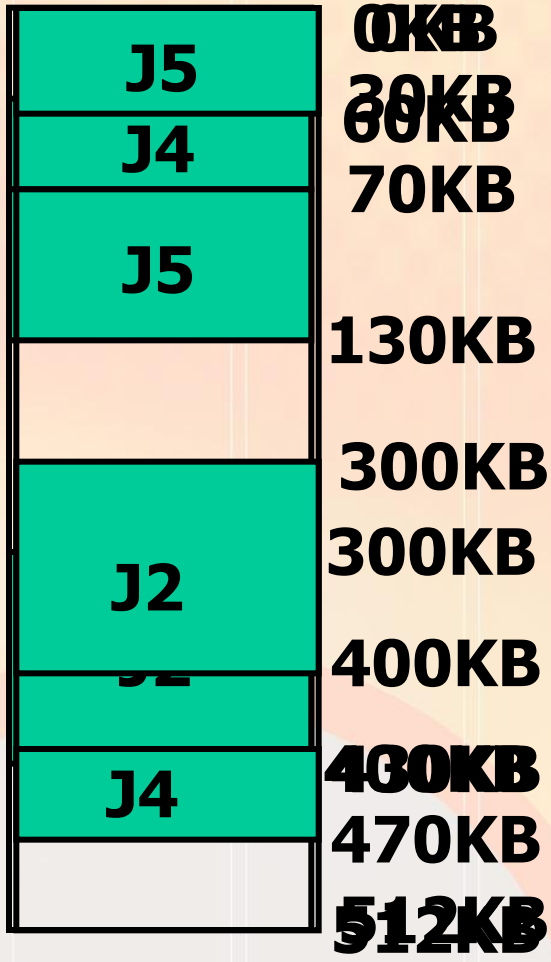
回收区与插入点的后一空闲分区**F2**相邻接



回收区同时与插入点的前后两个分区邻接

- 例：某系统采用动态分区分配方式管理内存，用户区主存空间为512k，在内存分配时，系统优先使用空闲区低端地址。对于如下申请序列，分别画图表示使用首次适应算法和最佳适应算法进行内存分配和回收后，内存的最终映像图。

  J1 req (300KB)
  J2 req (100KB)
  J1 release (300KB)
  J3 req (30KB)
  J4 req (40KB)
  J3 release (30KB)
  J5 req (60KB)



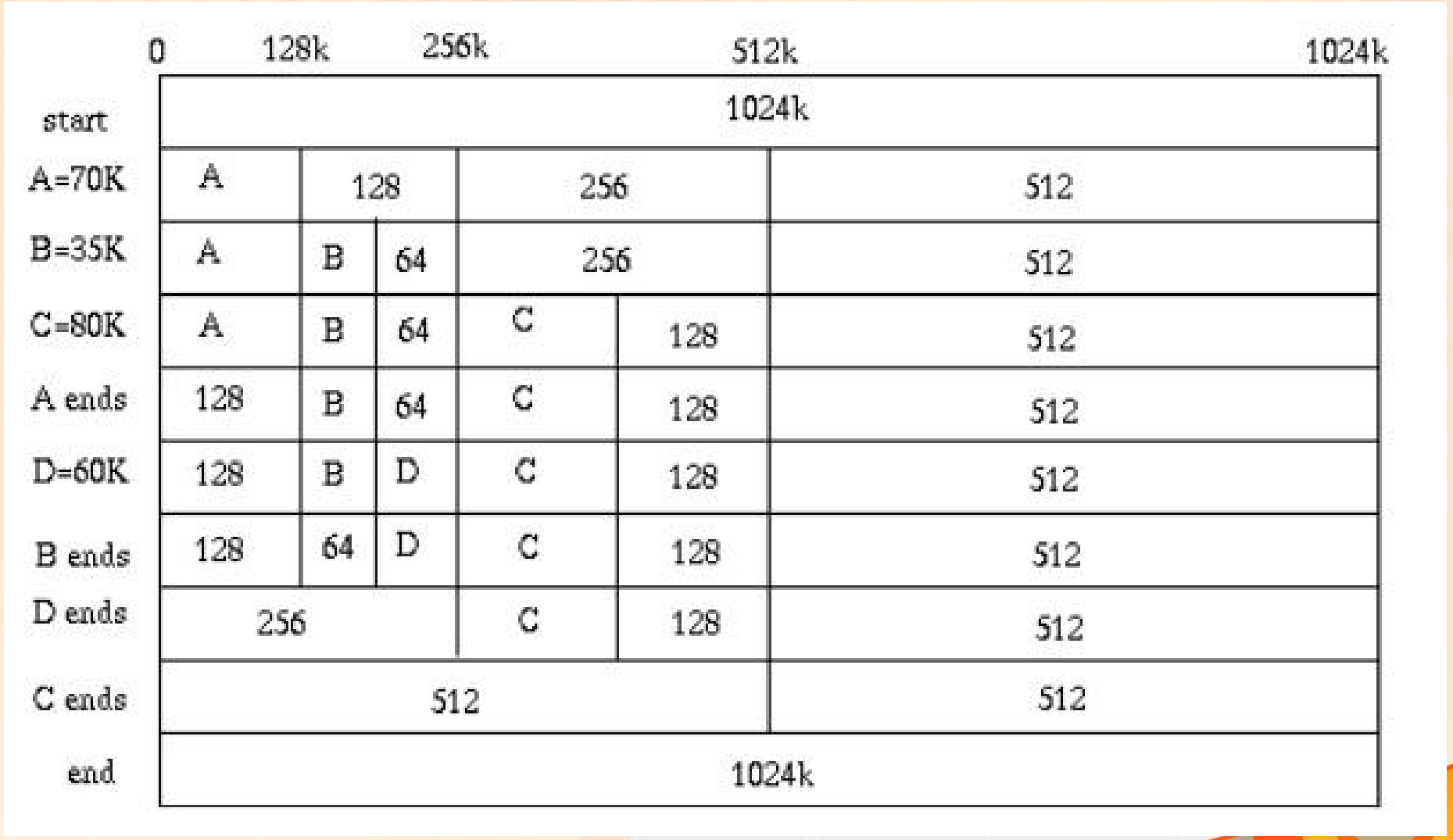
最佳适应算法

4.3.4 伙伴系统

- 伙伴系统方式是动态分区分配和固定分区分配的一种折衷方案。
- 伙伴系统规定，分配的分区和空闲分区大小都是2^k，k为整数，$1 \leqslant k \leqslant m$。$2^l$表示分配的最小分区大小，$2^m$表示分配的最大分区大小。$2^m$是整个可分配的内存大小。系统中也要建立一种管理索引表，指明每个链表表头。

- 分区分配方法：
  - 开始时，整个分区是2^m，在系统运行过程中，由于不断划分，可能会形成若干不连续的空闲分区，将它们分类，每一类具有相同大小，且每类建立一个空闲分区双向链表，系统中有若干个双向链表。
  - 当需要为进程分配大小为n的区块时，首先计算一个i，使$2^{i-1} \leqslant n \leqslant 2^i$，然后在大小为$2^i$的空闲分区链表中查找。
- 分区回收：
  - 若回收大小为2^i的分区，若有伙伴分区，则合并为2^{i+1}的分区，进而可能需要合并为2^{i+2}的分区……
- 算法性能取决于查找空闲分区的位置和分割、回收空闲分区所花费的时间。

例


0
128k
256k
512k
1024k
start
1024k
A=70K
A
128
256
512
B=35K
A
B
64
256
512
C=80K
A
B
64
C
128
512
A ends
128
B
64
C
128
512
D=60K
128
B
D
C
128
512
B ends
128
64
D
C
128
512
D ends
256
C
128
512
C ends
512
512
end
1024k

- 优点：
  - 快速搜索合并
  - 低外部碎片
- 缺点：
  - 内部碎片。
  - 因为按2的幂划分块，如果碰上66单位大小，那么必须划分128单位大小的块。

4.3.5 可重定位分区分配

- 当内存驻留多个进程时，分配一个区后大部分情况下都是有剩余零头的，因此在一个新作业到达时，就有可能零头分区的总和超过新作业要求的分区，但每一个空闲分区的容量都不够。

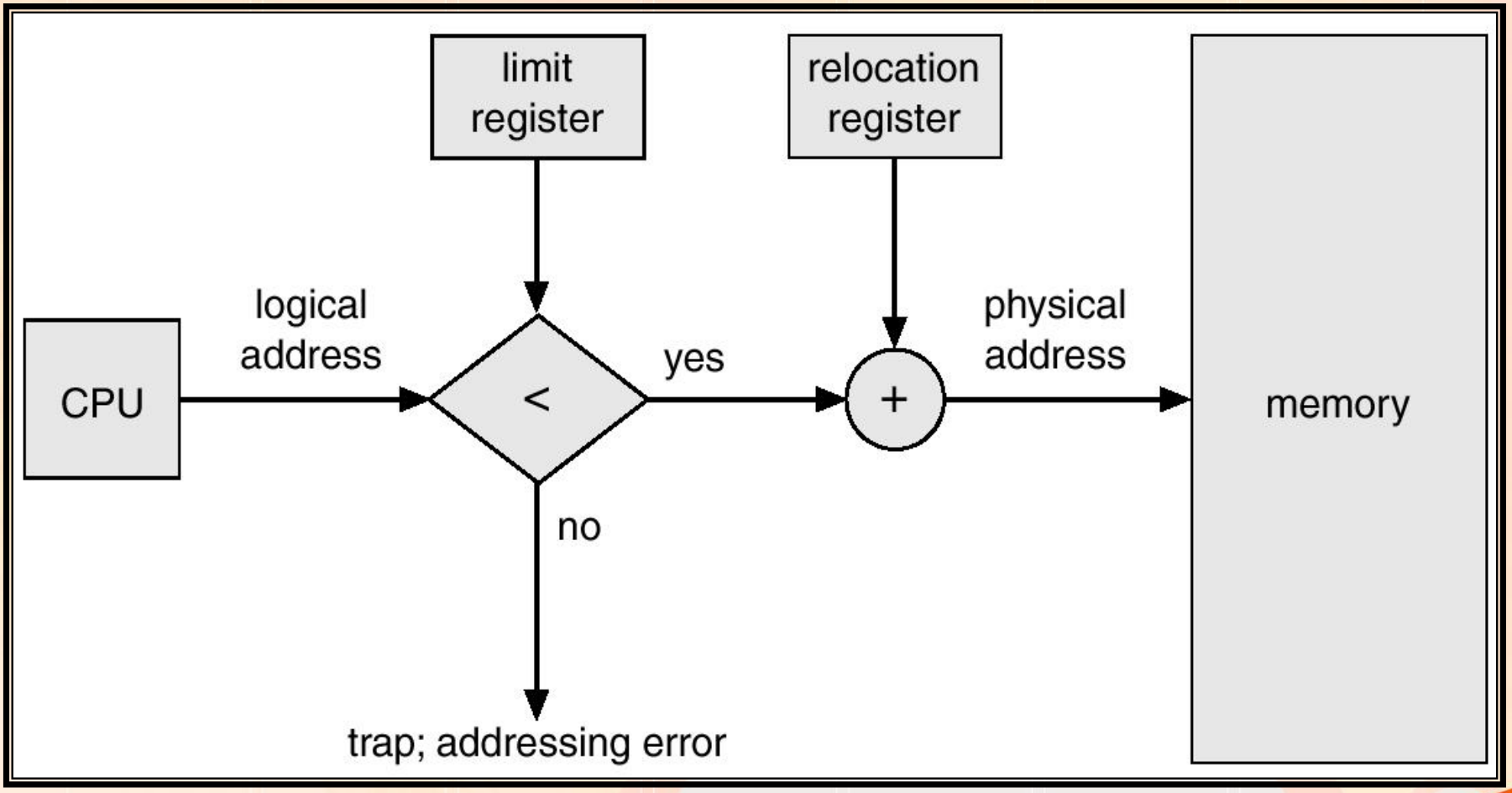
4.3.5 可重定位分区分配

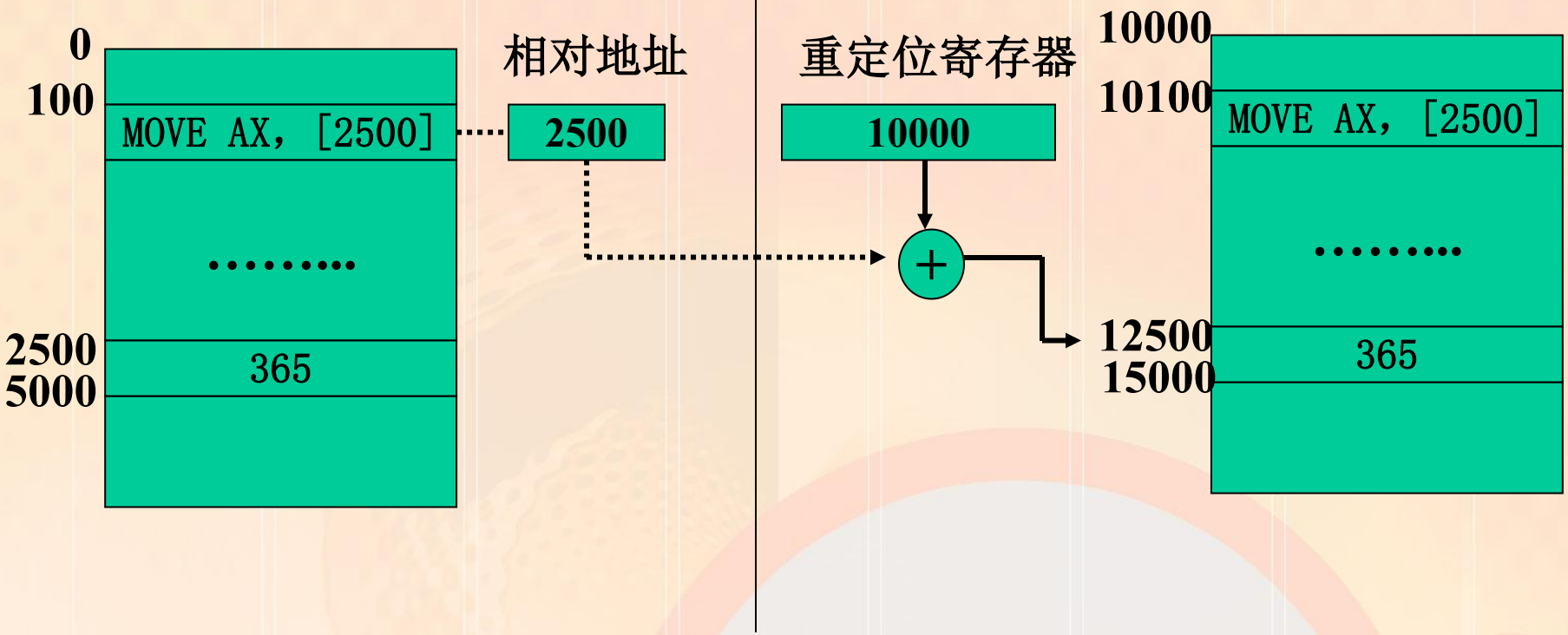
- 1. 紧凑(compaction)
  - 将各个占用分区向内存一端移动。使各个空闲分区聚集在另一端，合并为一个较大的空闲分区。

4.3.5　可重定位分区分配

- 对占用分区进行内存数据搬移占用CPU时间；如果对占用分区中的程序进行“浮动”，则其重定位需要硬件支持（重定位寄存器）。
- 何时执行紧凑操作：每个分区释放后，或内存分配找不到满足条件的空闲分区时

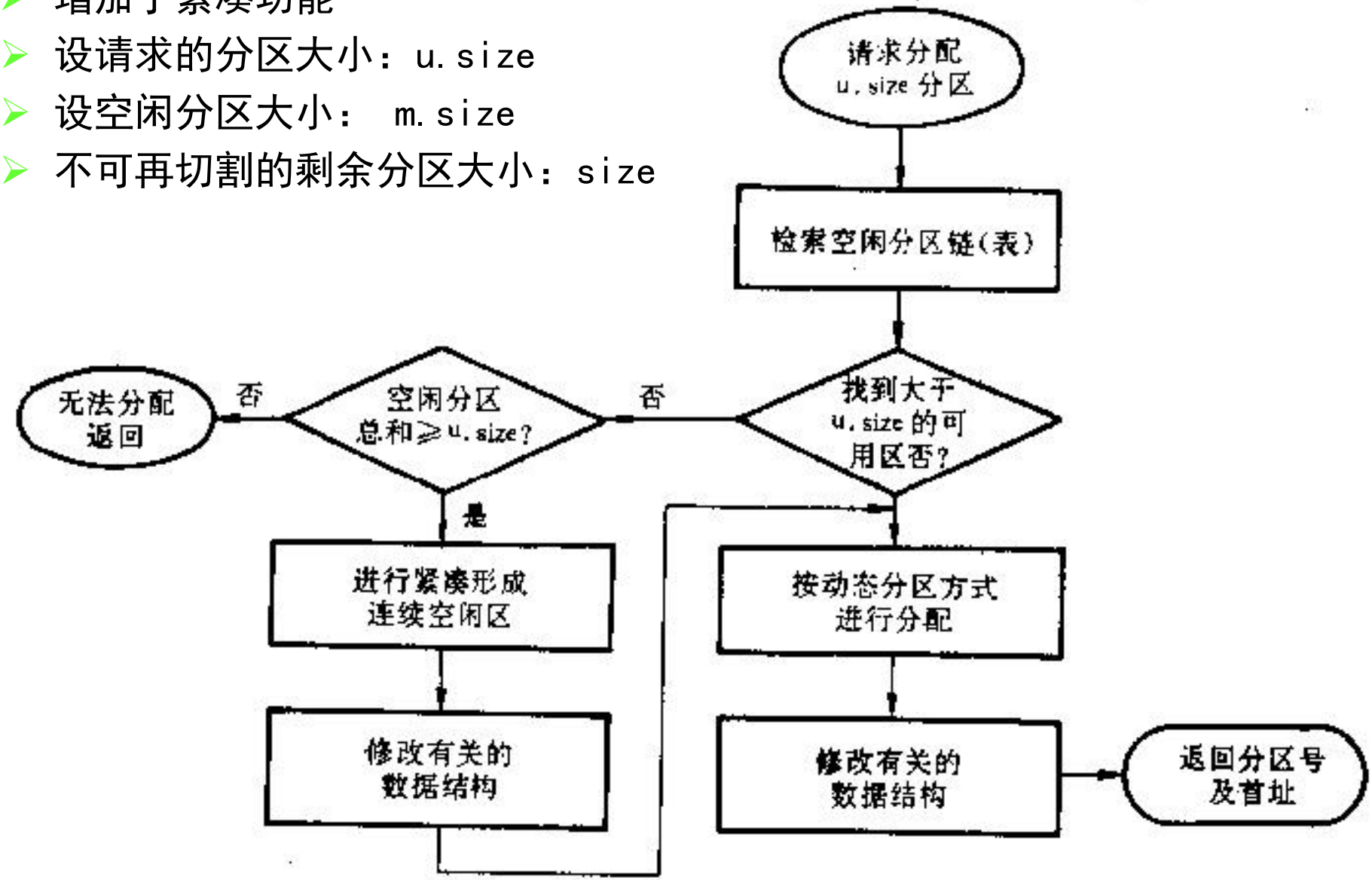
动态重定位


limit
register
relocation
register
logical
address
CPU
<
yes
+
physical
address
memory
no
trap; addressing error


0
100
MOVE AX，[2500]
2500
5000
365
相对地址
2500
重定位寄存器
10000
+
10000
10100
MOVE AX，[2500]
12500
15000
365

动态重定位分区分配算法

- 增加了紧凑功能
- 设请求的分区大小：u.size
- 设空闲分区大小： m.size
- 不可再切割的剩余分区大小：size

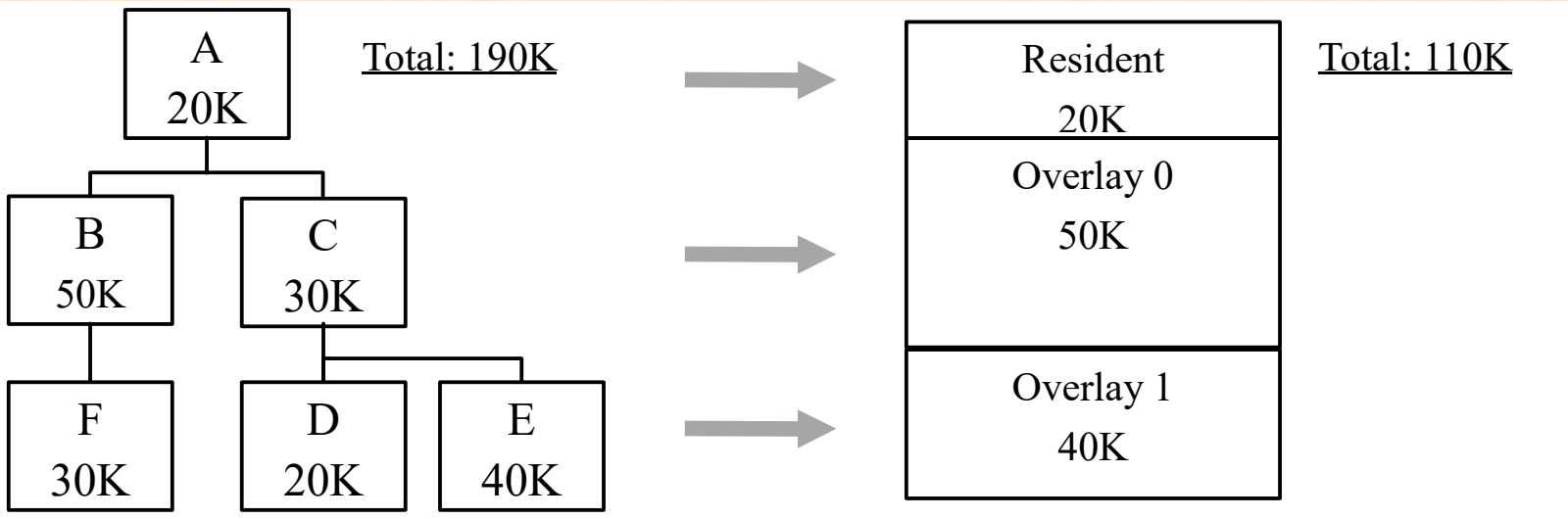


4.3.6 覆盖(overlay)

- 引入
  - 目标是在较小的可用内存中运行较大的程序。常用于多道程序系统，与固定分区存储管理配合使用。
- 原理：
  - 一个程序的几个代码段或数据段，按照时间先后来占用同一内存空间。
    - 将程序的必要部分（常用功能）的代码和数据常驻内存；
    - 可选部分（不常用功能）在其他程序模块中实现，平时存放在外存中（覆盖文件），在要用到时才装入到内存；
    - 彼此不存在调用关系的模块不必同时装入到内存，从而可以相互覆盖。

4.3.6 覆盖(overlay)

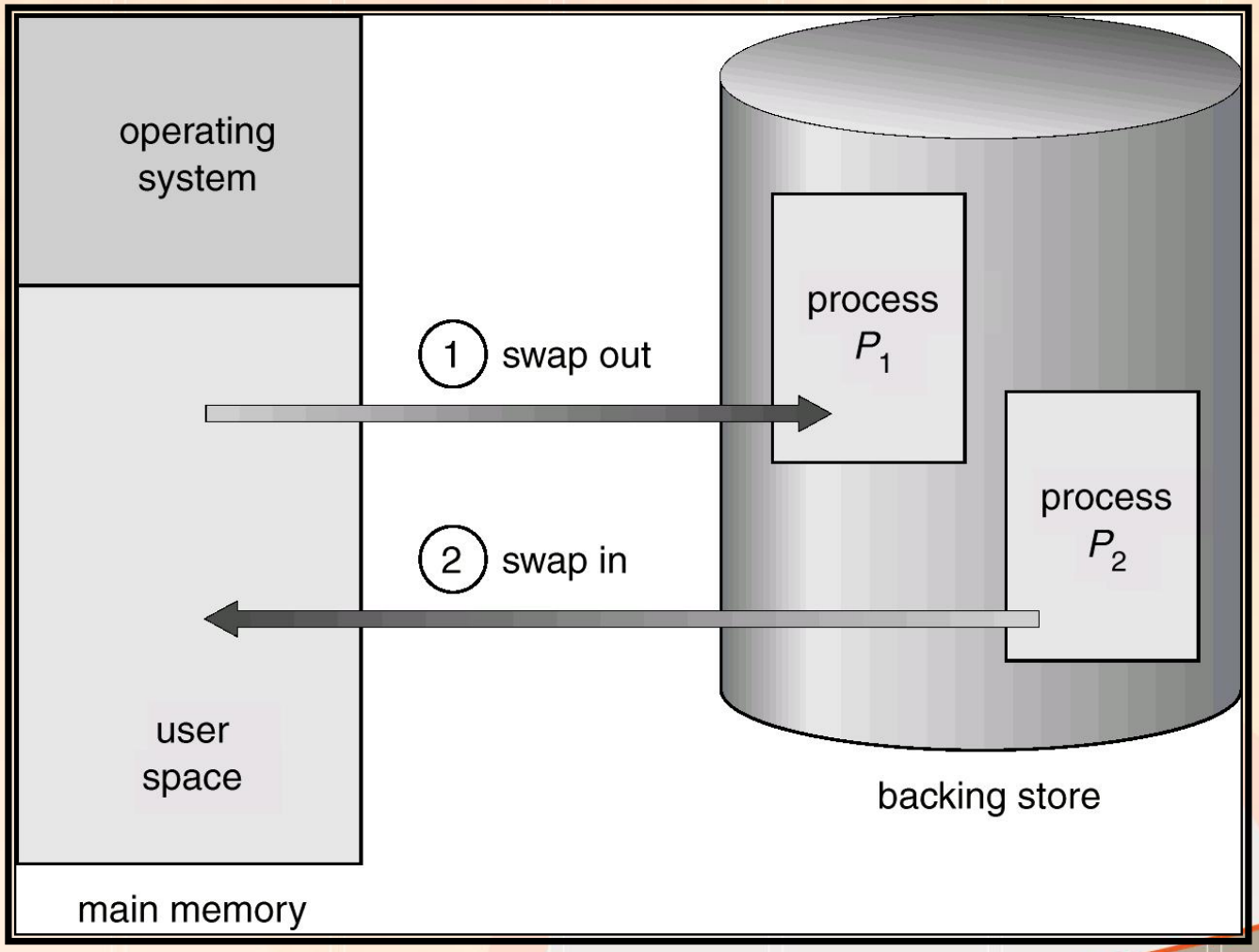
- 缺点：
  - 编程时程序员必须划分程序模块并确定程序模块之间的覆盖关系，增加了编程复杂度。
  - 进程在执行过程中要从外存装入覆盖文件，速度慢，以时间来换取空间。



4.3.7 对换(swapping)

- 对换的引入
  - 在多道程序环境下，一方面内存中有的进程处于阻塞态，无法执行；另一方面又有就绪进程在外存等待。所以引入对换。
  - 对换是将暂时不能执行的程序或数据送到外存中，从而获得空闲内存空间来装入具备运行条件的进程或进程所需要的程序和数据。
  - 进程暂时不能执行的可能原因：处于阻塞状态，低优先级（确保高优先级进程执行）
- 交换单位为整个进程的地址空间。
- 常用于多道程序系统或小型分时系统中，与可重定位分区分配存储管理配合使用。又称作“滚进/滚出(roll-in/roll-out)”。

对换


operating system
process P_1
1 swap out
process P_2
2 swap in
user space
backing store
main memory

4.3.7 对换(swapping)

- 对换空间的管理
  - 在具有对换功能的OS中，外存被分为对换区和文件区。
  - OS对对换区管理的目标是加快进程换入、换出的速度，因此采取连续分配方式，较少考虑碎片问题。

对换空间的管理

一般从磁盘上划出一块空间作为对换区使用

	存储内容	驻留时间	主要目标	分配方式
文件区	文件	较长久	提高文件存储空间的利用率	离散
对换区	从内存换出的进程	短暂	提高进程换入和换出的速度	连续

在系统中设置相应的数据结构以记录外存的使用情况。

对换空间的分配与回收，与动态分区方式时的内存分配与回收雷同。

进程的换入与换出

换出：当前执行进程需要更多内存，或系统创建了一个高优先级进程，而无内存空间时，OS选择处于阻塞态且优先级低的进程传送到磁盘的对换区，修改PCB。回收内存空间。

换入：OS定期查看进程的状态，在内存有空时，找出就绪且换出时间最久的进程换入内存。

优缺点

优点：增加并发运行的进程数目，并且给用户提供适当的响应时间；提高系统吞吐率。

缺点：对换入和换出的控制增加处理机开销；进程的整个地址空间都进行传送，没有考虑执行过程中地址访问的统计特性。

- 覆盖与交换比较
  - 与覆盖技术相比，交换不要求程序员给出程序段之间的覆盖结构。
  - 交换主要是在进程或作业之间进行，而覆盖则主要在同一作业或进程内进行。另外覆盖只能覆盖那些与覆盖程序段无关的程序段。

4.5 基本分页存储管理方式

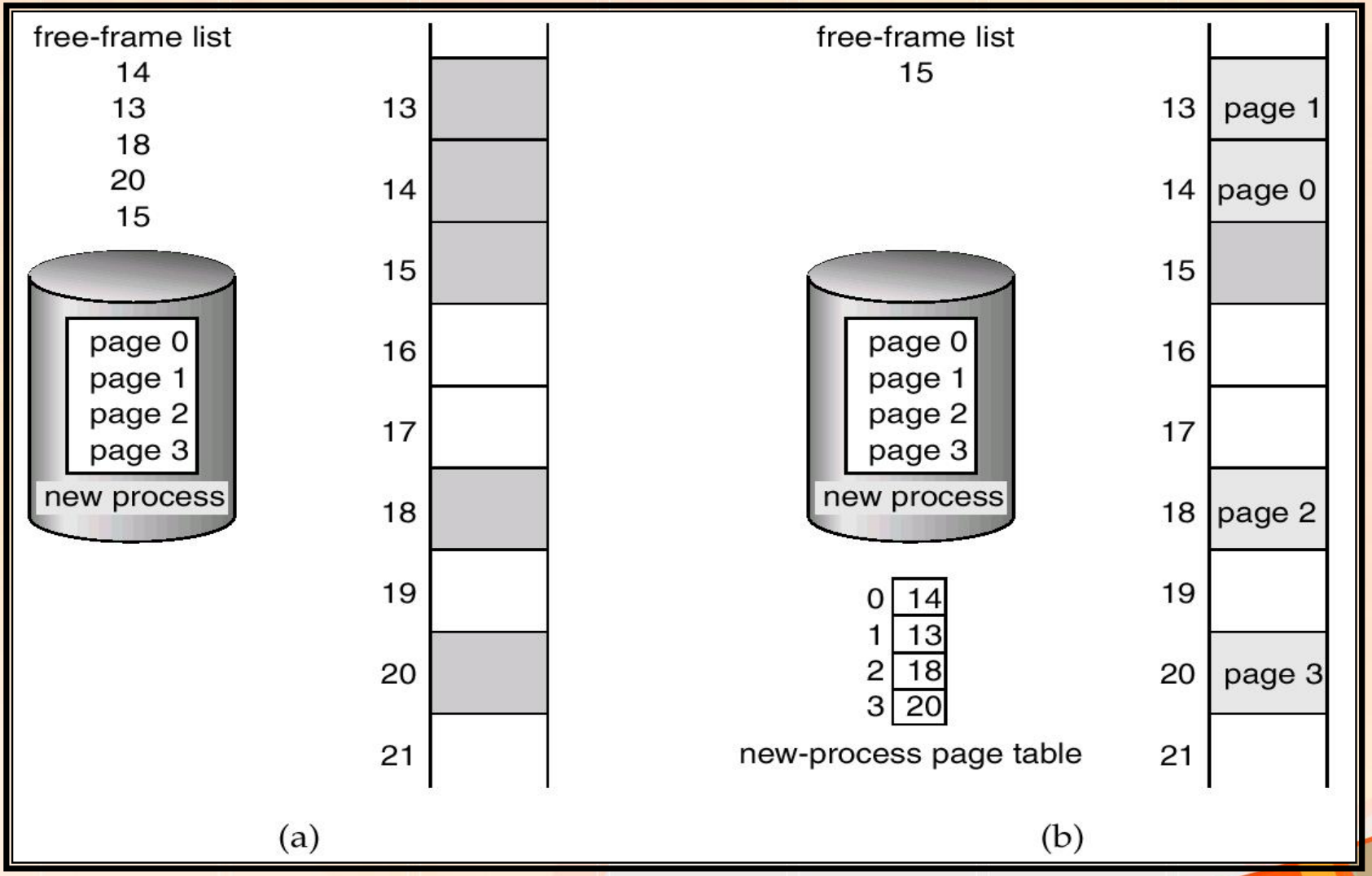
- 连续分配的问题：
  - 形成许多碎片：内碎片和外碎片
  - 紧凑带来开销。
- 离散分配方式
  - 若离散分配的基本单位是页，则称为分页存储管理；
  - 若离散分配的基本单位是段，则称为分段存储管理。
  - 基本分页存储管理不支持虚存技术，要求把整个作业都装入内存，才能运行。

- 在页式管理中：
  - 物理内存被划分为固定大小的页框(page frame)，也叫页帧，物理页框。
  - 进程的逻辑地址空间也分成同样大小的页(Page)。程序加载时，分配其所需的全部页，这些页不必连续。
  - 固定：一个计算机系统的内存容量是固定的，一个页的容量在硬件设计时也是确定的。

4.5 基本分页存储管理方式

- 进程装载
  - 在装入一个进程时，需找空闲页框，OS要将这些页框分配给装入的进程，进程地址空间的每个页占用一个页框，进程占用的所有页框不要求连续。
  - 要解决逻辑地址到物理地址的映象，需硬件支持。

如何为进程分配物理页框



Before allocation

After allocation

4.5 基本分页存储管理方式

- 基本分页管理中的数据结构
  - 进程页表：每个进程有一个页表，描述该进程的每个逻辑页占用的物理页框号。
  - 物理页面表：整个系统有一个物理页面表，描述所有物理页框的分配使用状况。数据结构：位示图，空闲页面链表；
  - 请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换；
  - 请求表也可以结合到各进程的PCB里，此时在PCB中记录本进程页表所在的物理页框号。上下文切换时，由OS将其加载到页表寄存器中。

4.5 基本分页存储管理方式

- 逻辑地址结构
  - CPU执行指令时产生的逻辑地址被分为两部分：

逻辑页号	页内偏移量
m-n位	n位

  - 逻辑地址空间为2^m，页框大小为2^n(m>n)，则逻辑地址的高m-n位为逻辑页号，低n位为页内偏移量。

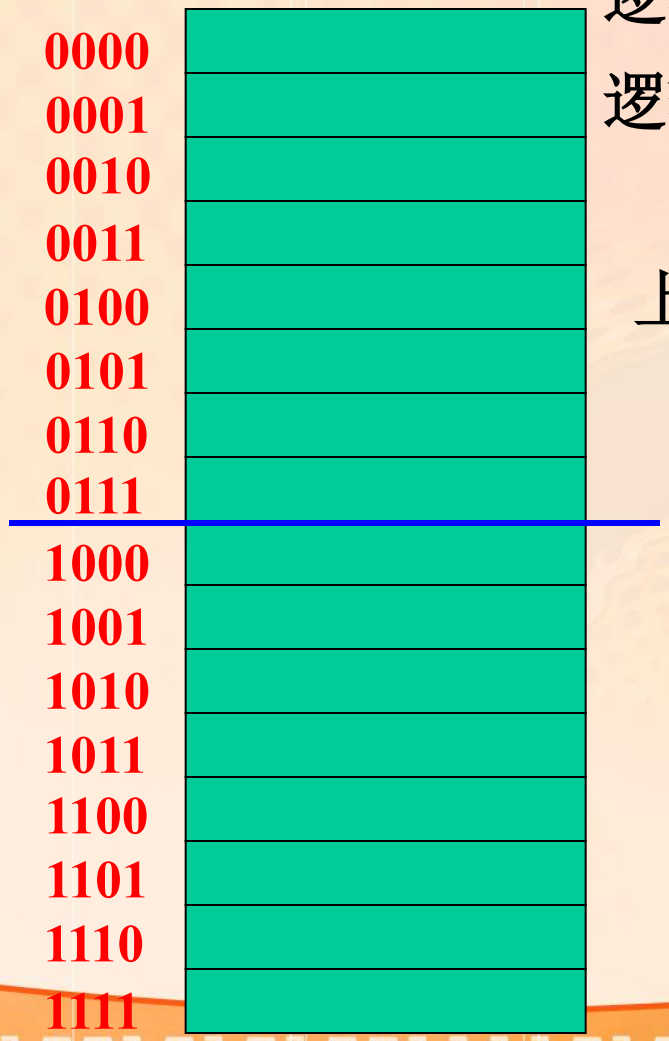
逻辑页号=INT [A/ L]

页内偏移量=[A] MOD L

A：逻辑地址　　L：页面尺寸

4.5 基本分页存储管理方式

设物理页框为8字节，16个字节的逻辑地址，逻辑上分成两组，称为两个逻辑页。



逻辑地址0100可理解为0页，页内地址为4

逻辑地址1101可理解为1页，页内地址为5

这样，一个物理上的一维地址在逻辑上成为了二维地址（页号，页内地址）

同样，设页框大小为4字节，则逻辑页号为高二位，00，01，10，11；
页内地址低二位，00，01，10，11

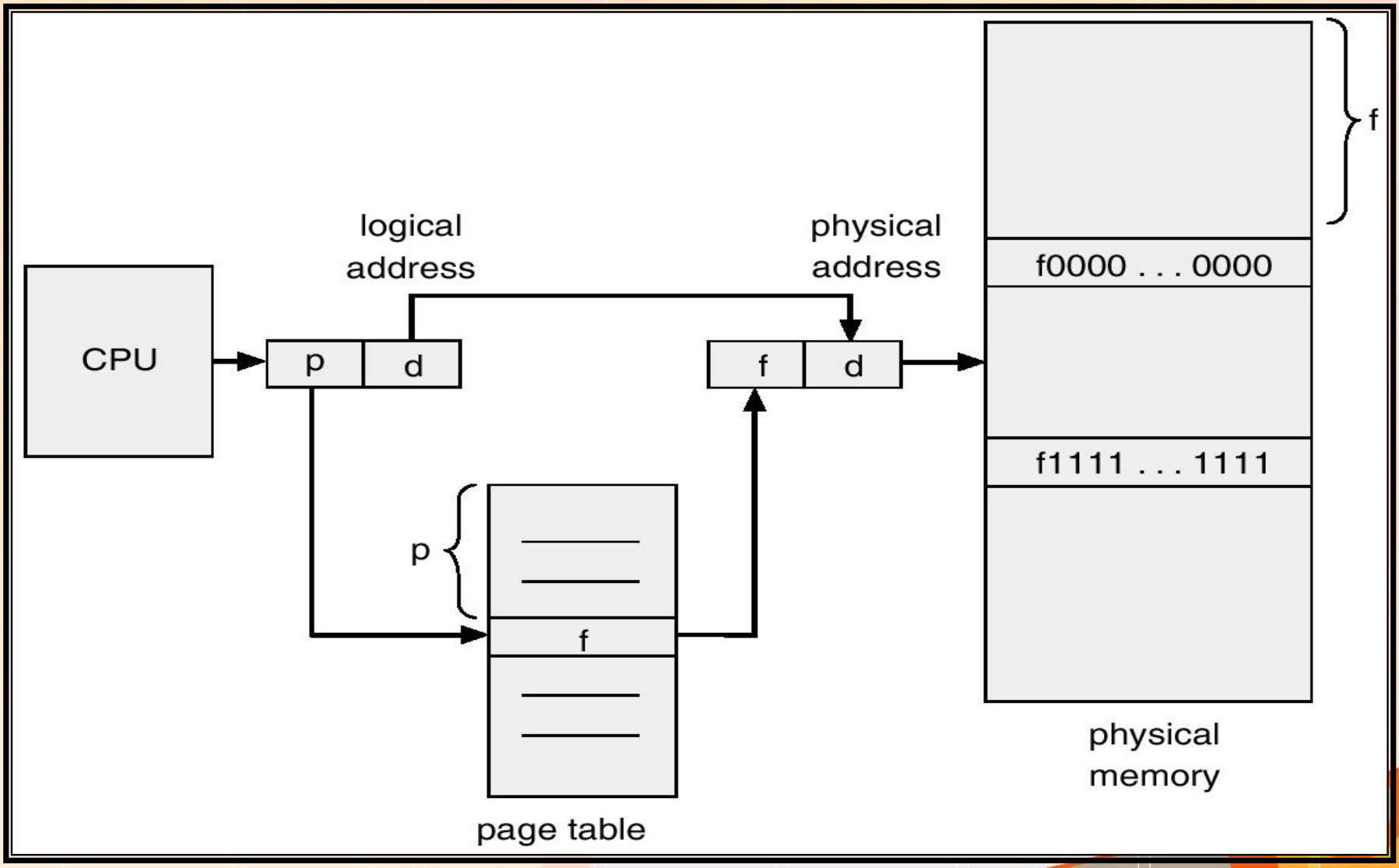
- 页面大小的选择
  - 和目前计算机的物理内存大小有关：2^n。
  - 较小的页面，减小内碎片，但加大页表的长度，从而形成新的开销并增加换入、换出的开销；
  - 较大的页面，减小页表的长度，加大内碎片；管理开销小，交换时对外存I/O效率高。
  - 两者的折中。

- 页式管理的优缺点
  - 优点：
    - 没有外碎片，每个内碎片不超过页大小。
    - 一个程序不必连续存放。便于改变程序占用空间的大小（主要指随着程序运行而动态生成的数据增多，要求地址空间相应增大，通常由系统调用完成而不是操作系统自动完成）。
  - 缺点：
    - 程序全部装入内存。

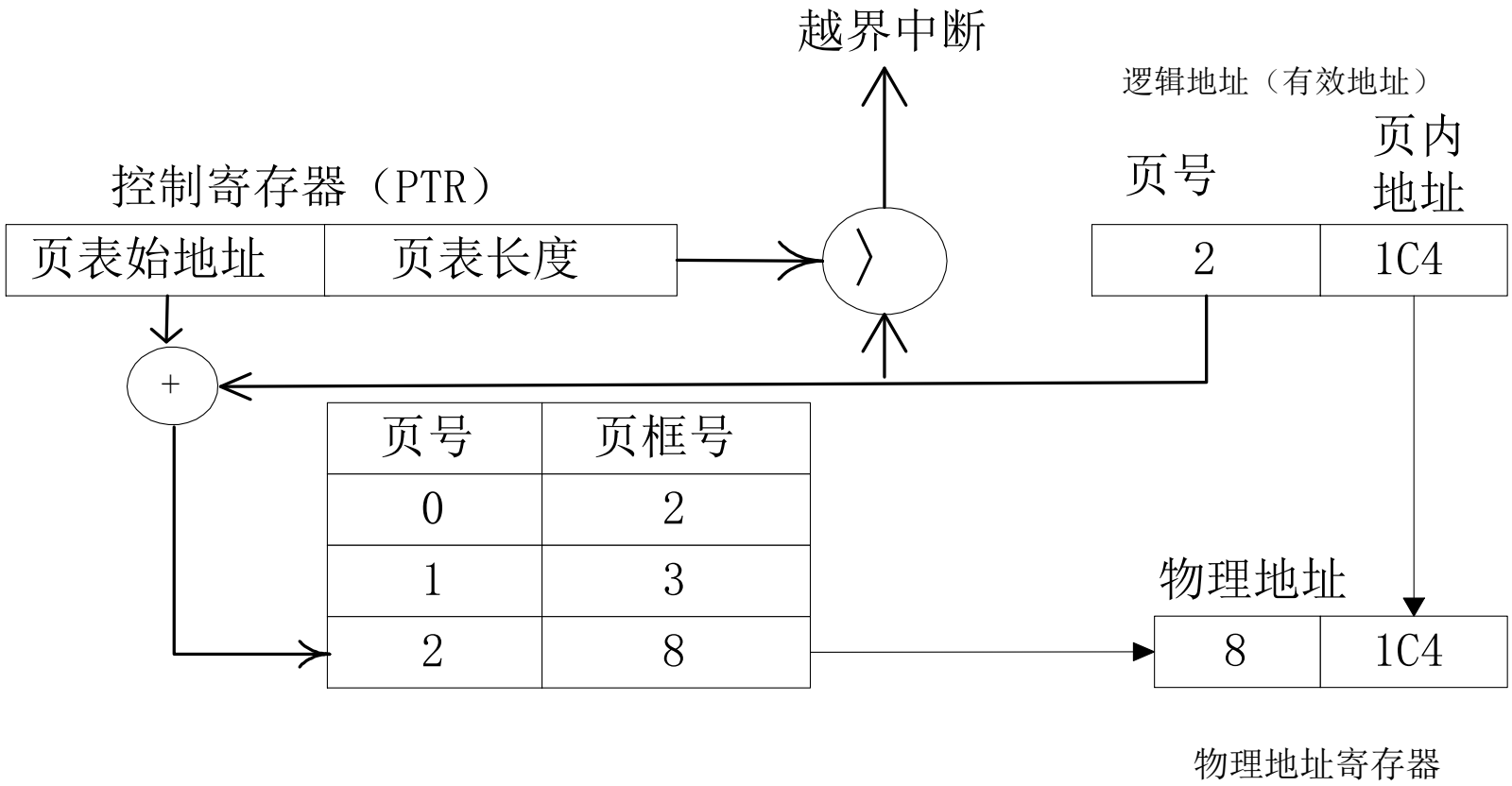
4.5.2　地址变换机构

- 基本地址变换机构
  - 逻辑上连续的目标程序在物理内存中已经不能保证连续存放，支持页式管理的机器硬件上都有一套地址变换机构完成逻辑地址到物理地址的变换。
  - 逻辑地址分为两部分：逻辑页号，页内偏移地址；
  - 通过查进程页表，得物理页号，从而形成物理地址。

4.5.2 地址变换机构


CPU
logical address
p
d
physical address
f
d
f
f0000 . . . 0000
f1111 . . . 1111
physical memory
p
f
page table

4.5.2　地址变换机构


越界中断
逻辑地址（有效地址）
页号
页内地址
控制寄存器（PTR）
页表始地址
页表长度
2
1C4
+
页号
页框号
0
2
1
3
2
8
物理地址
8
1C4
物理地址寄存器

4.5.2　地址变换机构

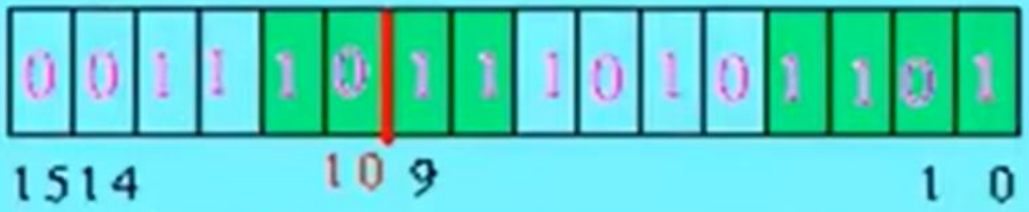
- 例题：关于分页的地址变换计算 -- 虚地址结构

（1）设页面大小为1KB，将逻辑地址划3BADH划分为页号和页内偏移量两部分。用16进制表示。

（2）设页面大小为2KB，将逻辑地址3BADH划分为页号和页内偏移量两部分。用16进制表示。

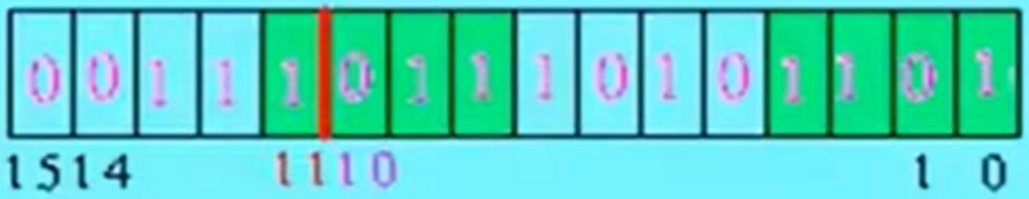
4.5.2 地址变换机构

例 1： 页面大小是 1KB， 虚地址是 3BADH



P=0EH W=3ADH

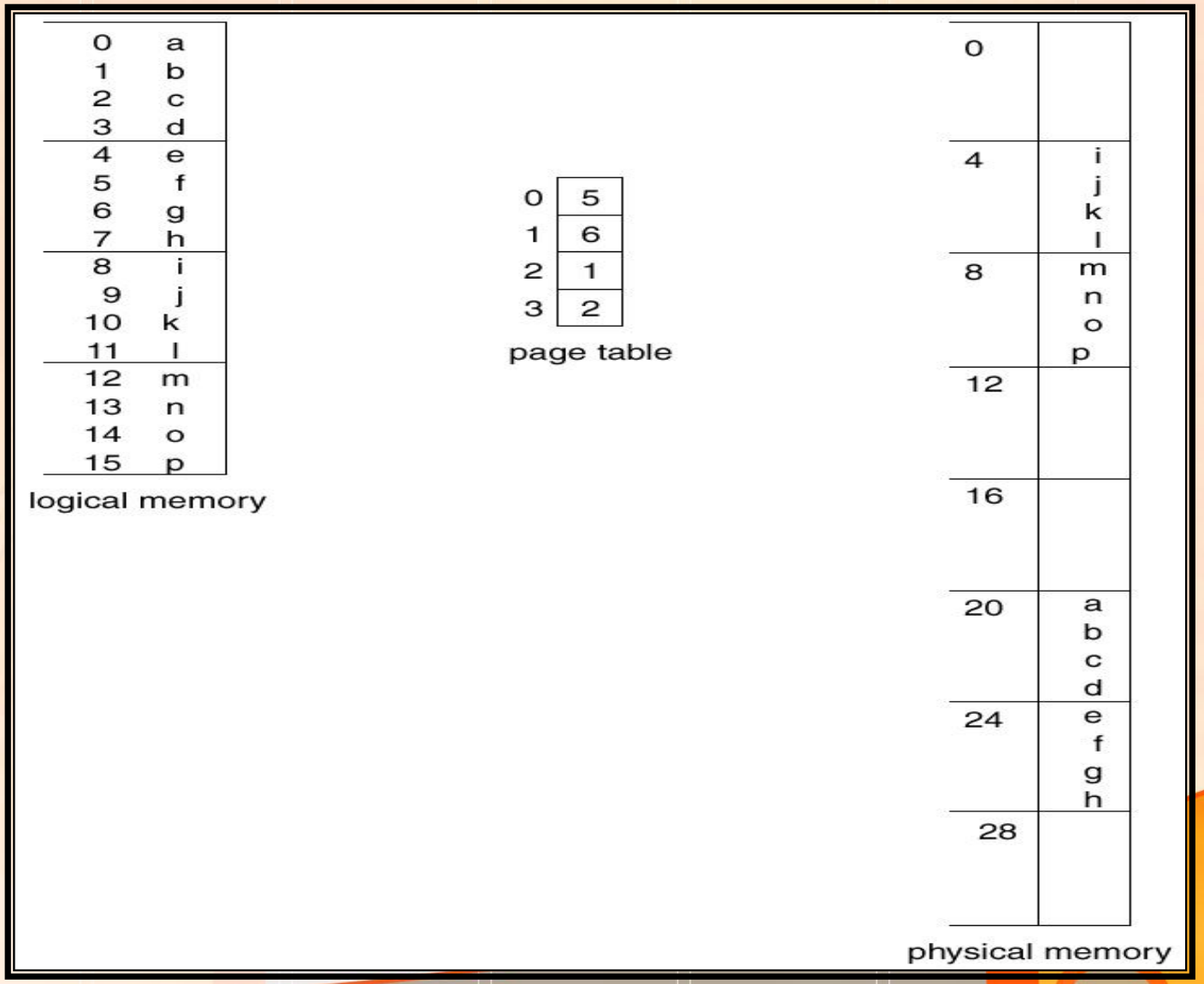
例 2： 页面大小是 2KB， 虚地址是 3BADH



P=07H W=3ADH

4.5.2 地址变换机构

- 例题：
- 页面大小为4 字节。物理内存的大小为32字节。对应于该进程页表，则逻辑地址0映象到物理地址20，逻辑地址13映象到物理地址9。



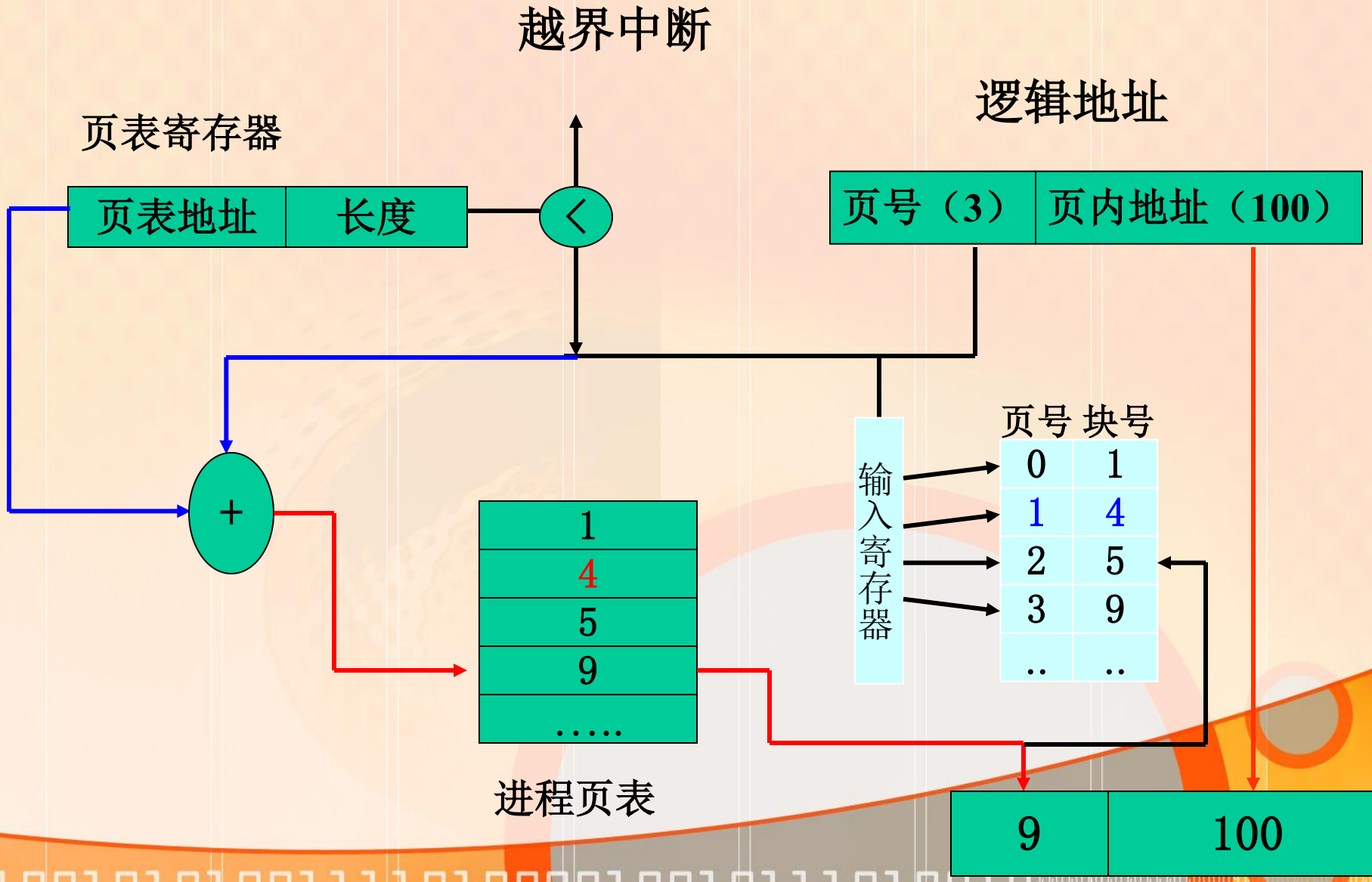
具有快表的地址变换机构

- 对绝大部分系统，页表是利用内存存储的，因此进行一次内存操作至少需要两次访问内存
  - 第一次读页表
  - 第二次访问数据。
- 如能将页表装在寄存器中访问就快得多，但如果将其全部放在寄存器中，则成本太大。

具有快表的地址变换机构

- 联想存储器：
  - 关联存储器Associative Memory
  - 具有并行查找功能的高速缓冲寄存器
  - 快表TLB
  - 根据程序局部性原理，将页表的一部分放在里面。
  - 个数一般在8到32个。（超过32个效果并不明显）
- 有快表的地址变换过程：
  - 访存时首先查找快表，若命中，则直接产生物理地址。只需访存一次。
  - 若快表不命中，则查找内存里的页表，并按照一定的置换算法，置换快表中的页表项。生成物理地址，此时要访存两次。

具有快表的地址变换机构



- 例：在页式存储管理中，假定访问主存的时间为200毫微秒，访问高速缓冲存储器的时间为40毫微秒，高速缓冲存储器为16个单元，查快表的命中率为90%，则将逻辑地址转换成绝对地址进行存取的平均时间为___
- (40+200+200)*10%+(40+200)*90%=260毫微秒

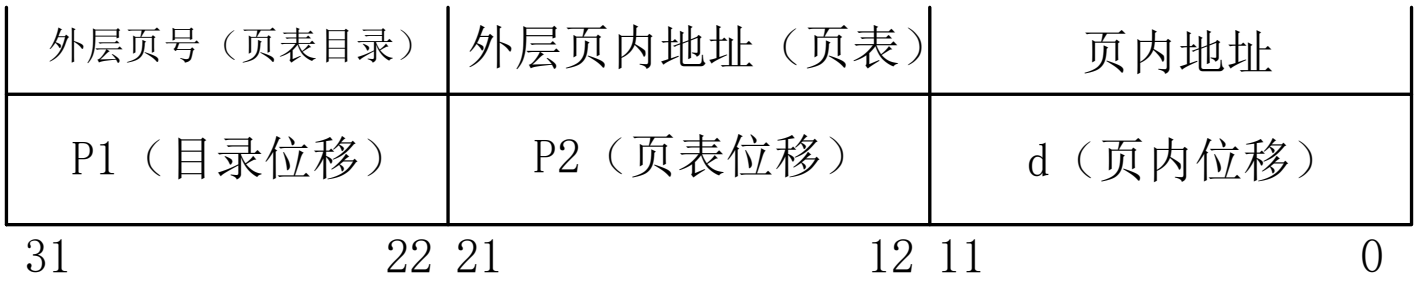
4.5.4　两级和多级页表

- 现代计算机系统，CPU都支持非常大的逻辑地址空间（2^{32}--2^{64}）。这样可想而知页表会非常大。
- 设逻辑地址宽度为32bit，假设页面大小为4K（即2^{12}），页表项达1M之多，每个页表项为4个字节，仅页表项就要占用4MB的连续内存空间。解决方法：
  - 分散存储；
  - 当前需要的放在内存，其余的暂存于磁盘。

4.5.4 两级和多级页表

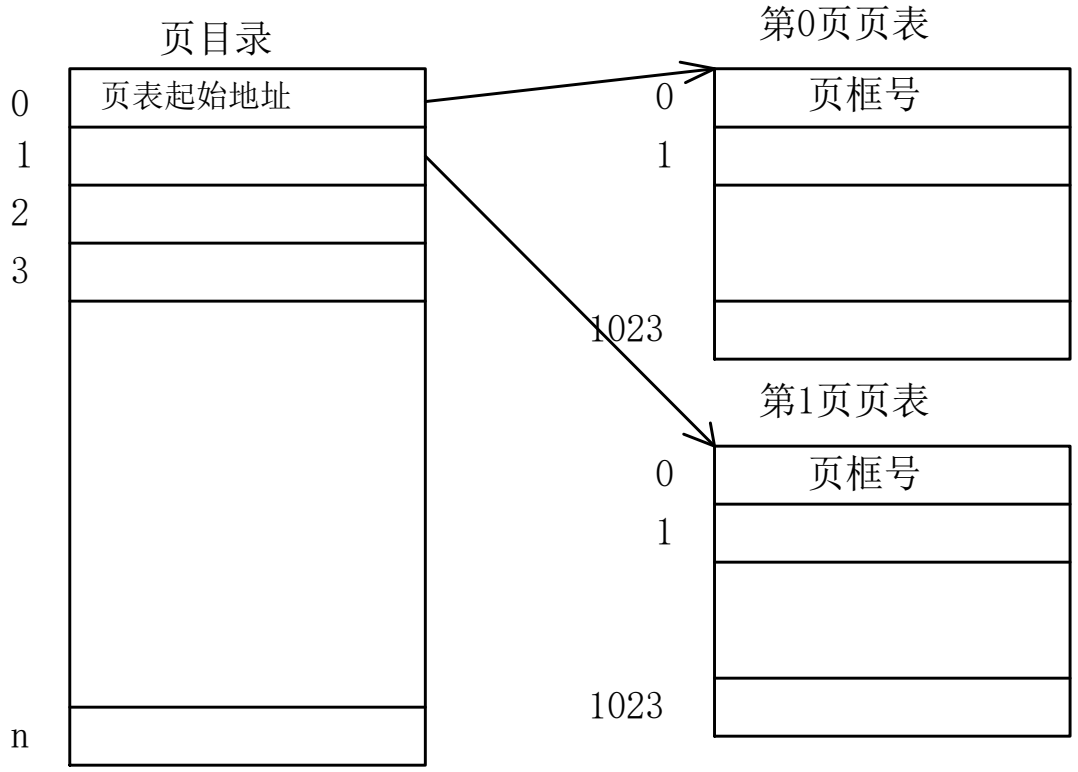
- 两级页表（Two Level Page Table）
  - 分散存储：将页表分页，每个页面的大小与物理页框的大小相同。解决难于找到大的连续物理内存的问题。
  - 相应的机制：增加页表的页表，即外层页表，也叫页目录表。
  - 页目录表也存放在内存中。此时，PCB中存放的是本进程对应的页目录表所在的物理页框号。上下文切换时，由OS加载到专用寄存器。

逻辑地址结构：

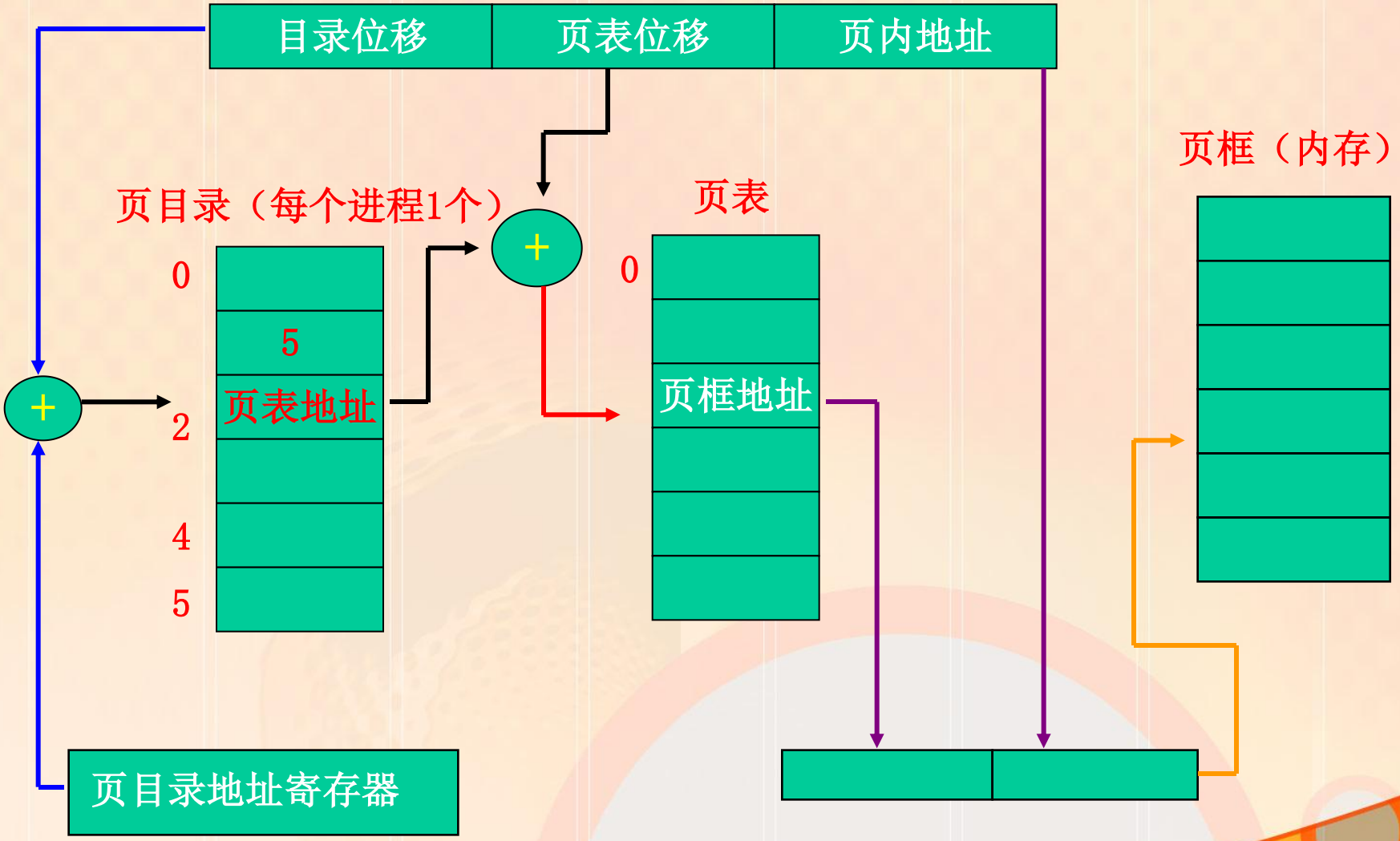


4.5.4　两级和多级页表

- 两级页表结构：



4.5.4 两级和多级页表



数据存取需通过3次访问内存

4.5.4 两级和多级页表

- 多级页表
  - 对于64位字长的机器，虚拟地址空间很大而每页比较小，则进程页表太长。宜采用两级或多级页表。例如SUN的SPARC处理器，支持3级页表；而Motorola的68030处理器甚至支持4级页表。
  - 为缩短查找时间，多级页表中的每级都可以装入到关联存储器（即页表的高速缓存）中，并按照cache的原理进行更新。

4.5.4 两级和多级页表

- 例：物理页框大小为4B，每个页表项占2B，某进程逻辑地址空间32B。
- 进程的代码段、数据段要占用8个页框，页表有8项。页表占用空间大小为16B，要分为4页存放。
- 故二级页表有4项，占用内存8B。又分为2页存放，所以一级页表有2项，正好可以放入一页中。

设逻辑地址为13，则对应的物理地址是_____

作业

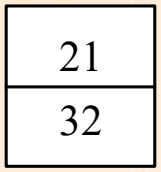
- 某计算机有32位虚地址空间，且页大小为1024字节。每个页表项长4个字节。因为每个页表都必须包含在一页中，所以使用多级页表，则

  （1）需要几级页表？

  （2）地址映象时，逻辑地址被分为几部分？每部分几位？

作业

- 某系统主存按字节编址，内存采用分页管理机制，物理页框大小为8B，每个页表项占2B。进程P的逻辑地址空间为64B，进程P的PCB中存放的最外层页表所在的物理页框号为8。图中给出了进程P各级页表依次存放的物理页框号（注：页表项的其他内容省略了）。请问：（1）P进程的页表分为几级？（2）计算进程P在执行时产生的逻辑地址20所对应的物理地址。



8 号页框内存放的页框号

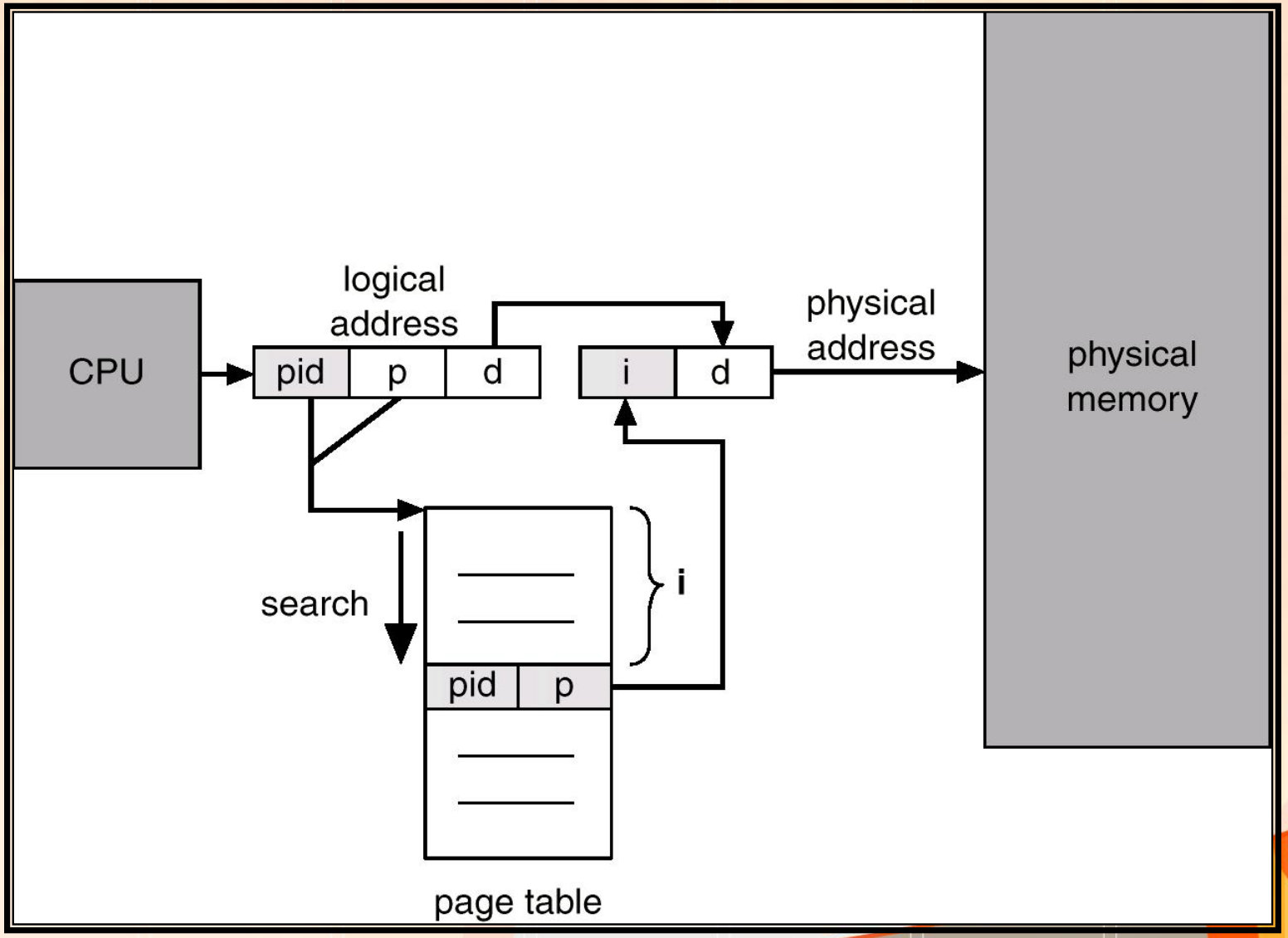
21 号页框内依次存放的页框号

32 号页框内依次存放的页框号

4.5.5 反置页表

- 反置页表（Inverted Page Table）
  - 以上方法，每个进程一张页表，页表按照进程的逻辑地址顺序排序，内容为物理块号（页框号）。
  - 反置页表则按物理块号的顺序排序，内容为隶属的进程id及其页号。
  - 实例：IBM AS/100、IBM RISC SYSTEM 6000等。
  - 在利用反置页表进行地址变换时，是利用进程id和页号，检索反置页表，实际上可利用联想存储器来检索

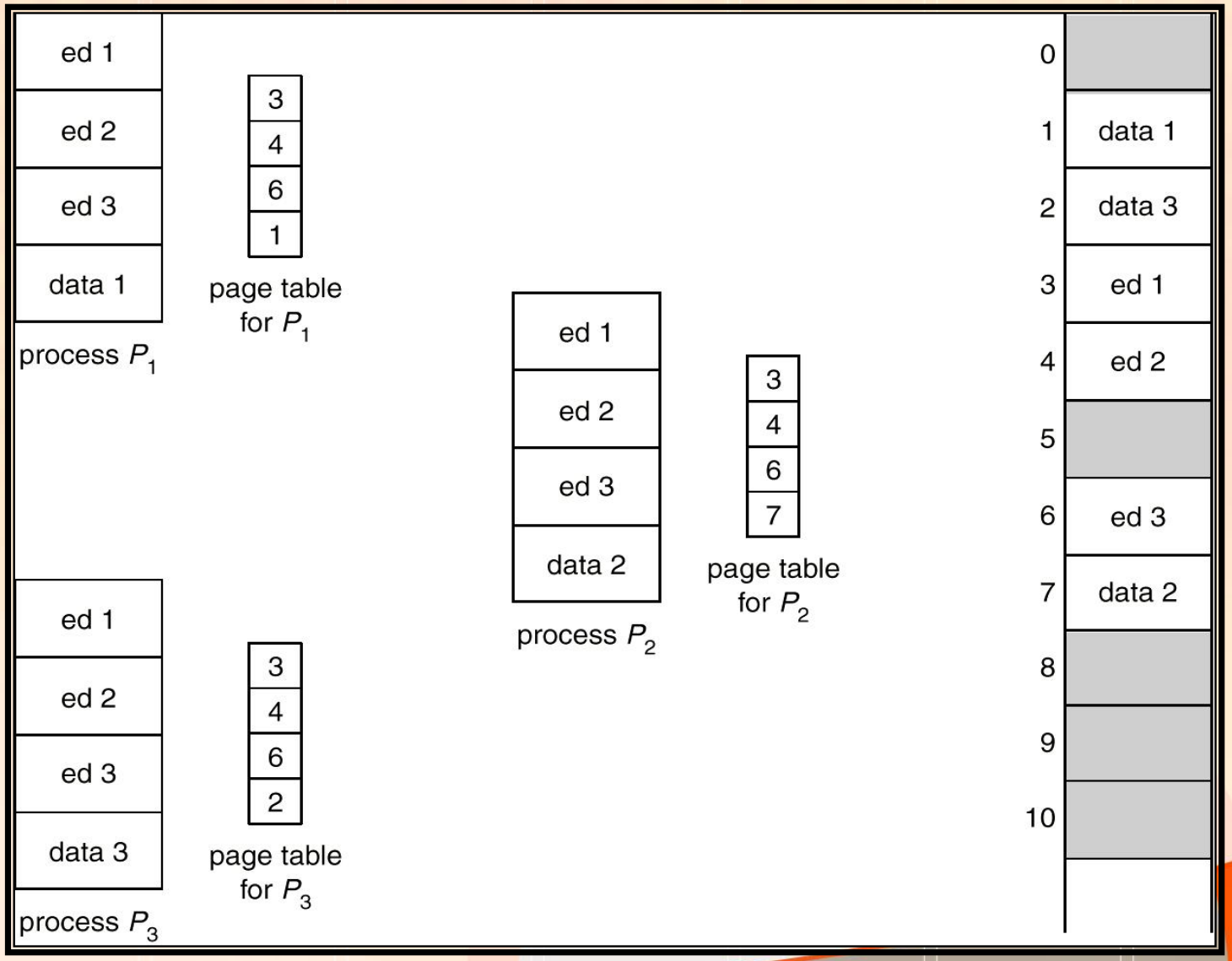
4.5.5 反置页表


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table

4.5.5 反置页表

- 采用反置页表技术，整个系统中只有一张页表，每个物理页框在页表中只有一项。
- 每次访存，查页表速度慢，可能需要查找整张表。
- 不易于实现页面共享。

页面共享


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10

- 在实际应用中，经常需要分配一组连续的页框，而频繁地申请和释放不同大小的连续页框，必然导致在已分配页框的内存块中分散了许多小块的空闲页框。这样，即使这些页框是空闲的，其他需要分配连续页框的应用也很难得到满足。
- 为了避免出现这种情况，Linux内核中引入了伙伴系统算法(buddy system)。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍。（页框大小为4KB）