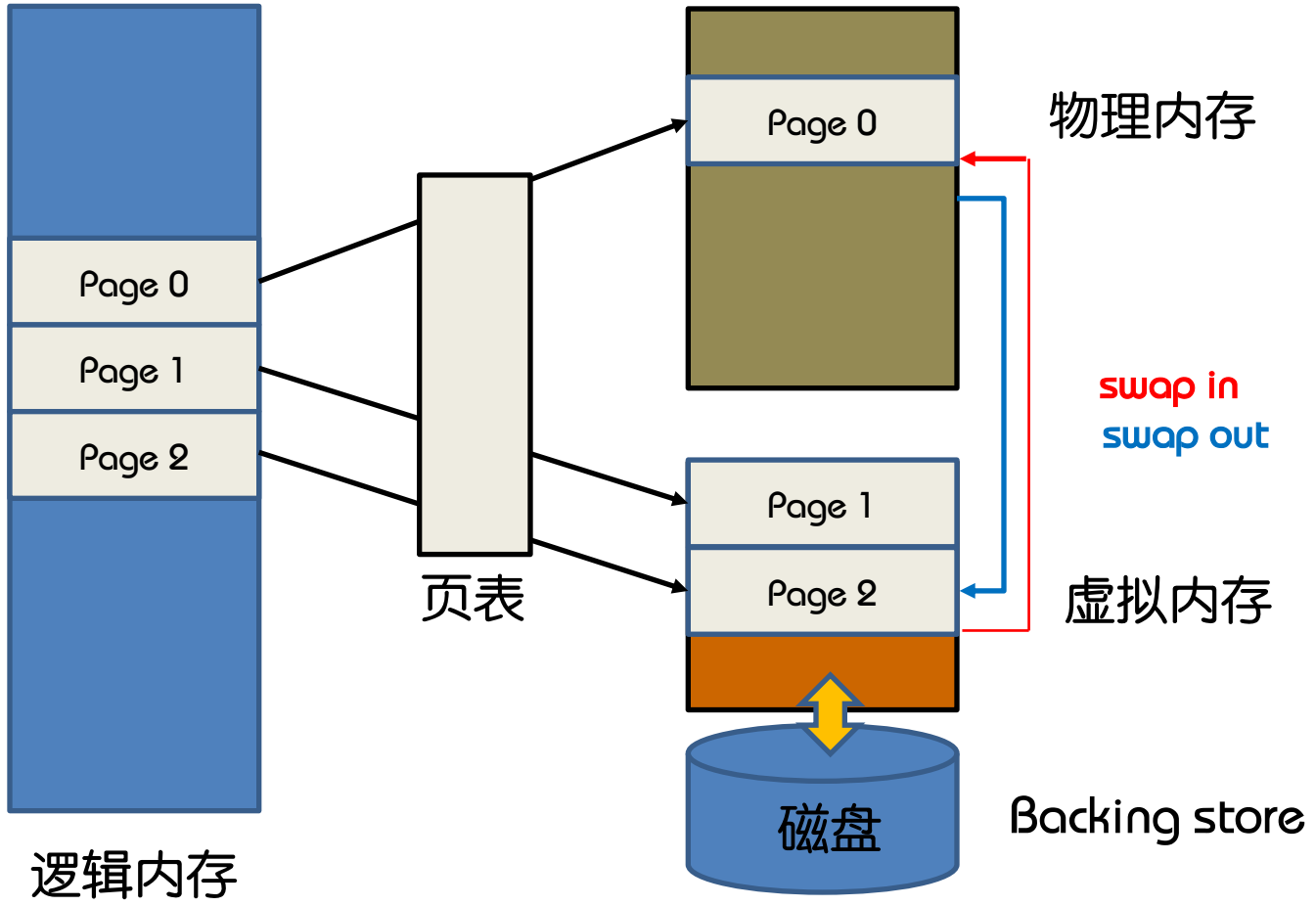
第九章、虚拟内存

1. 虚拟内存背景
2. 介绍按需调页、页错误、页置换
3. 页置换算法
4. 以及帧分配算法

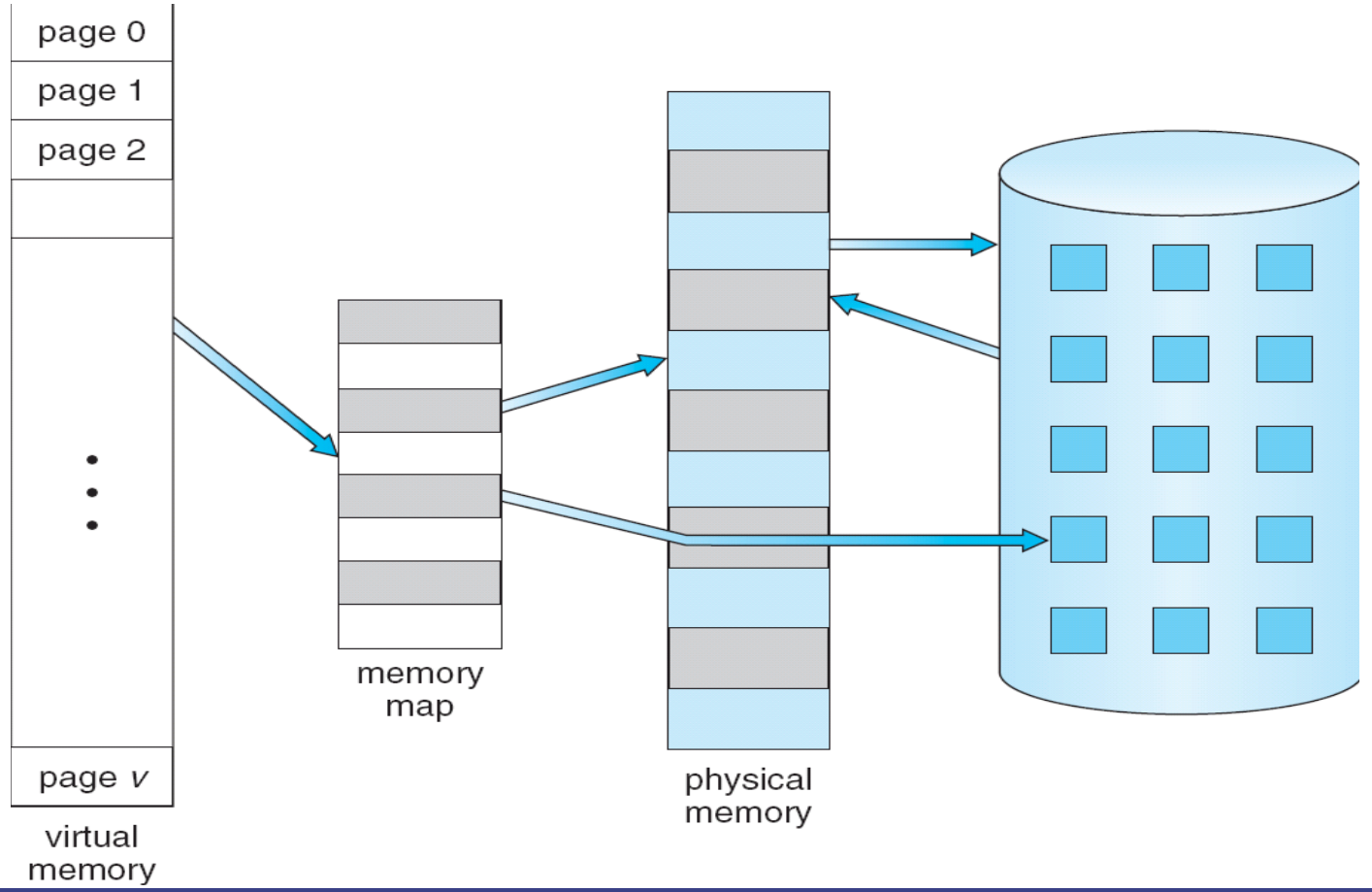
第一节、虚拟内存背景


Page 0
Page 1
Page 2
逻辑内存
页表
Page 0
Page 1
Page 2
物理内存
虚拟内存
swap in
swap out
磁盘
Backing store

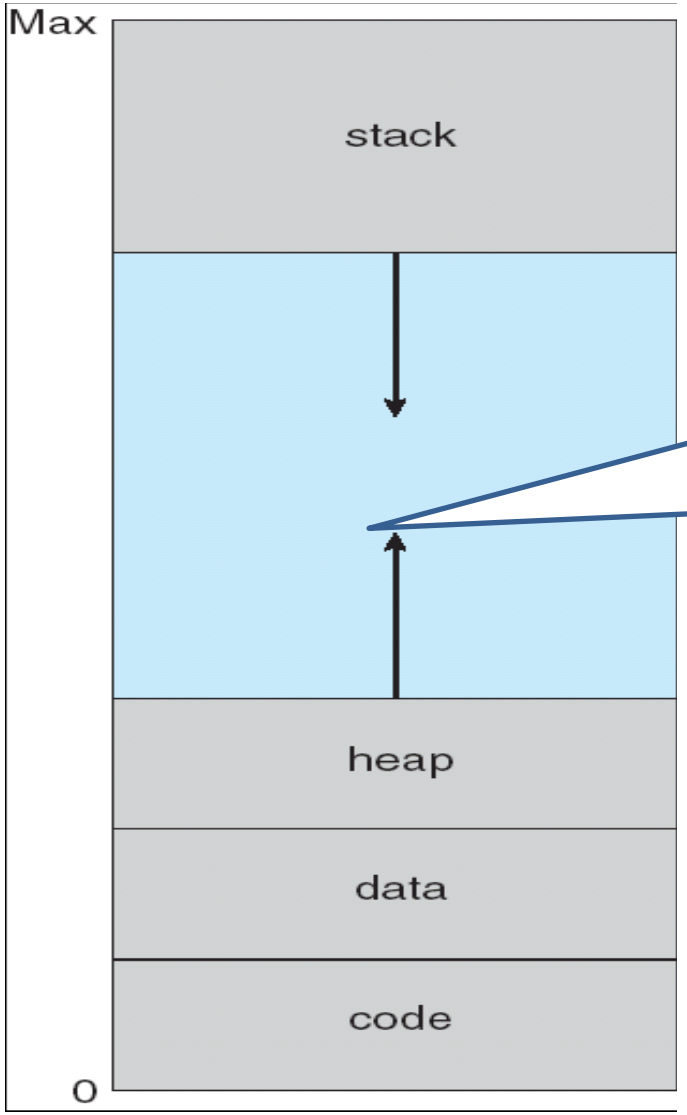
虚拟内存是内存管理的一种技术，它允许执行进程时不必完全载入内存，可以部分程序载入到内存

- 优点:
  I. 逻辑地址空间可大于物理地址空间
  II. 可以被多个进程共享地址空间
  III. 可以提供更有效的进程创建

进程的虚拟地址空间是如何在内存中存放的逻辑视图



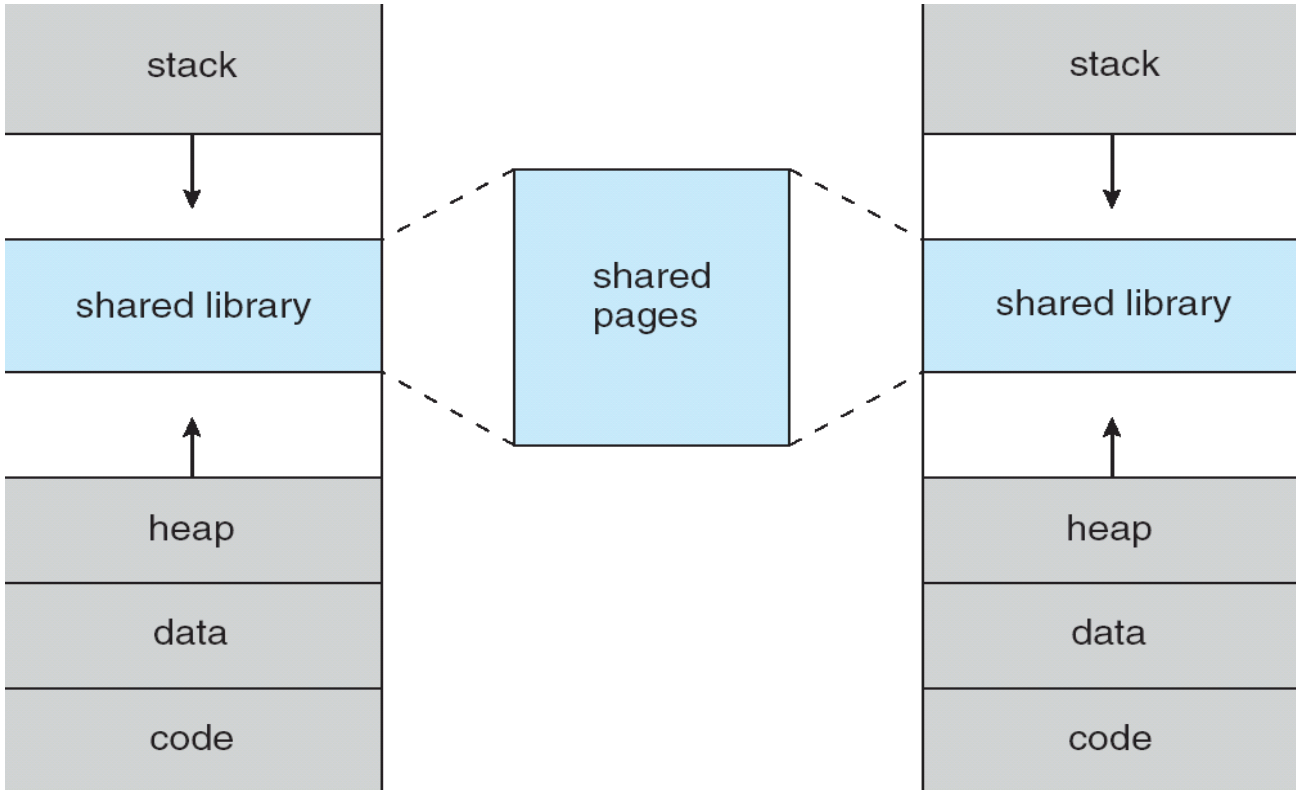
进程的虚拟地址空间



这个孔是虚拟地址的一部分，只有在堆和栈生长时，才需要实际的物理地址

使用虚拟内存的共享库

虚拟内存也允许共享，实现进程之间内存共享



第二节、按需调页、页错误、页置换

执行程序如何从磁盘载入内存（全部、部分）

- 为执行程序需要调入页、但暂时不需要的页不会调入到物理内存
  I. 可以减少 I/O
  II. 可以减少内存使用
  III. 应答时间快

那么，操作系统需要区分哪些页在物理内存，哪些页在虚拟内存

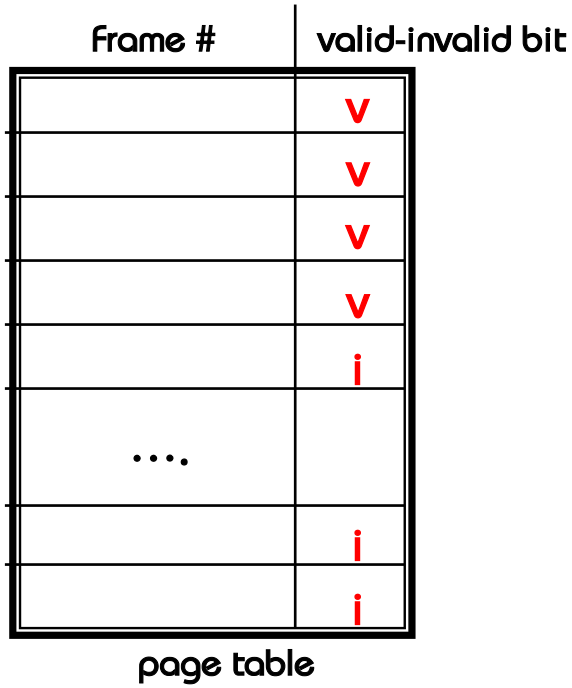
- 有效/无效位（valide/invalide）

有效-无效位

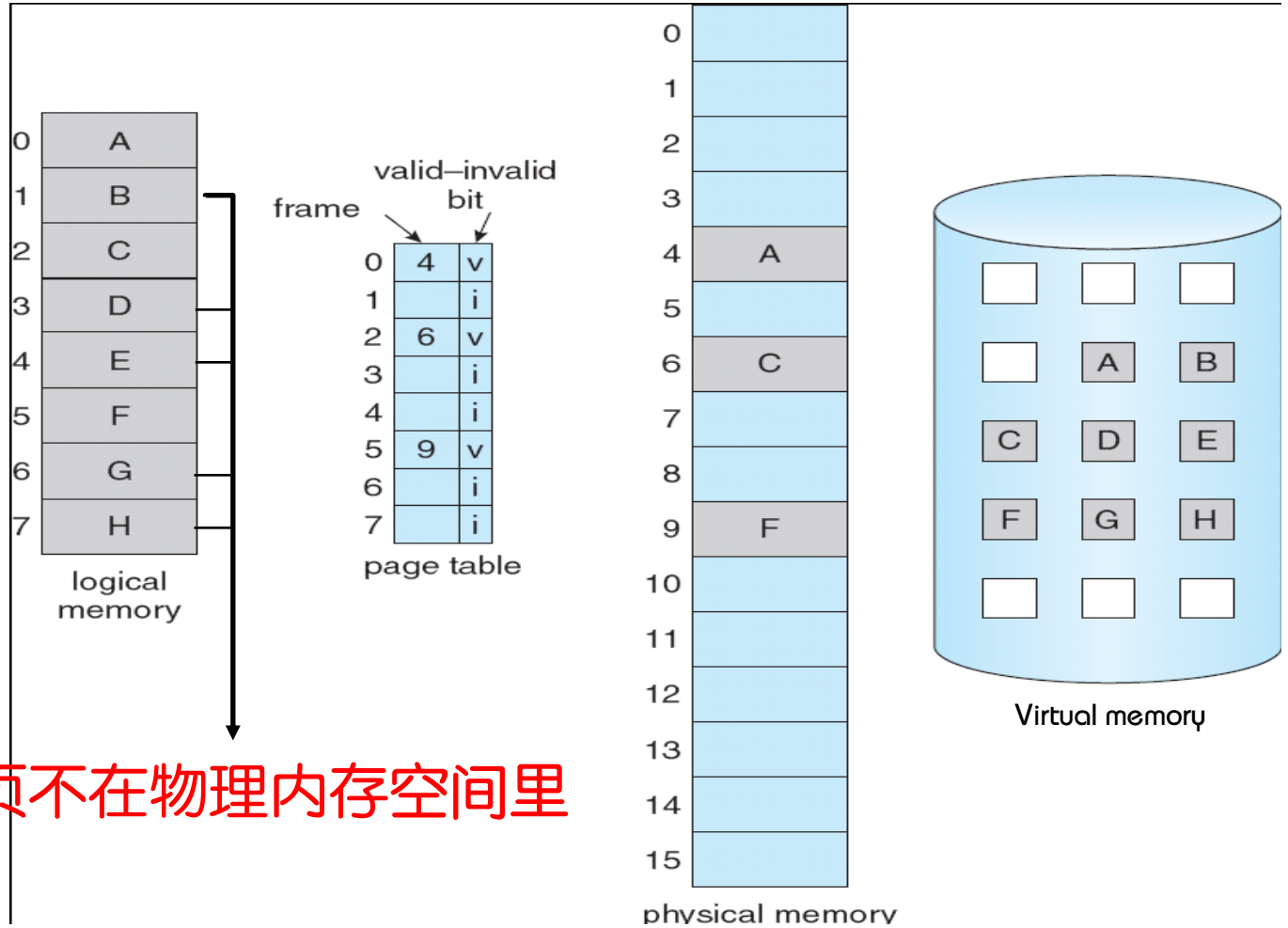
- 有效（v）:页在物理内存中
- 无效（i）:页不在物理内存中，即在虚拟内存中

I. 当访问无效页时，系统会出现页错误

II. 页错误触发载入（从虚拟内存到物理内存）



当有些页不在内存中时的页表



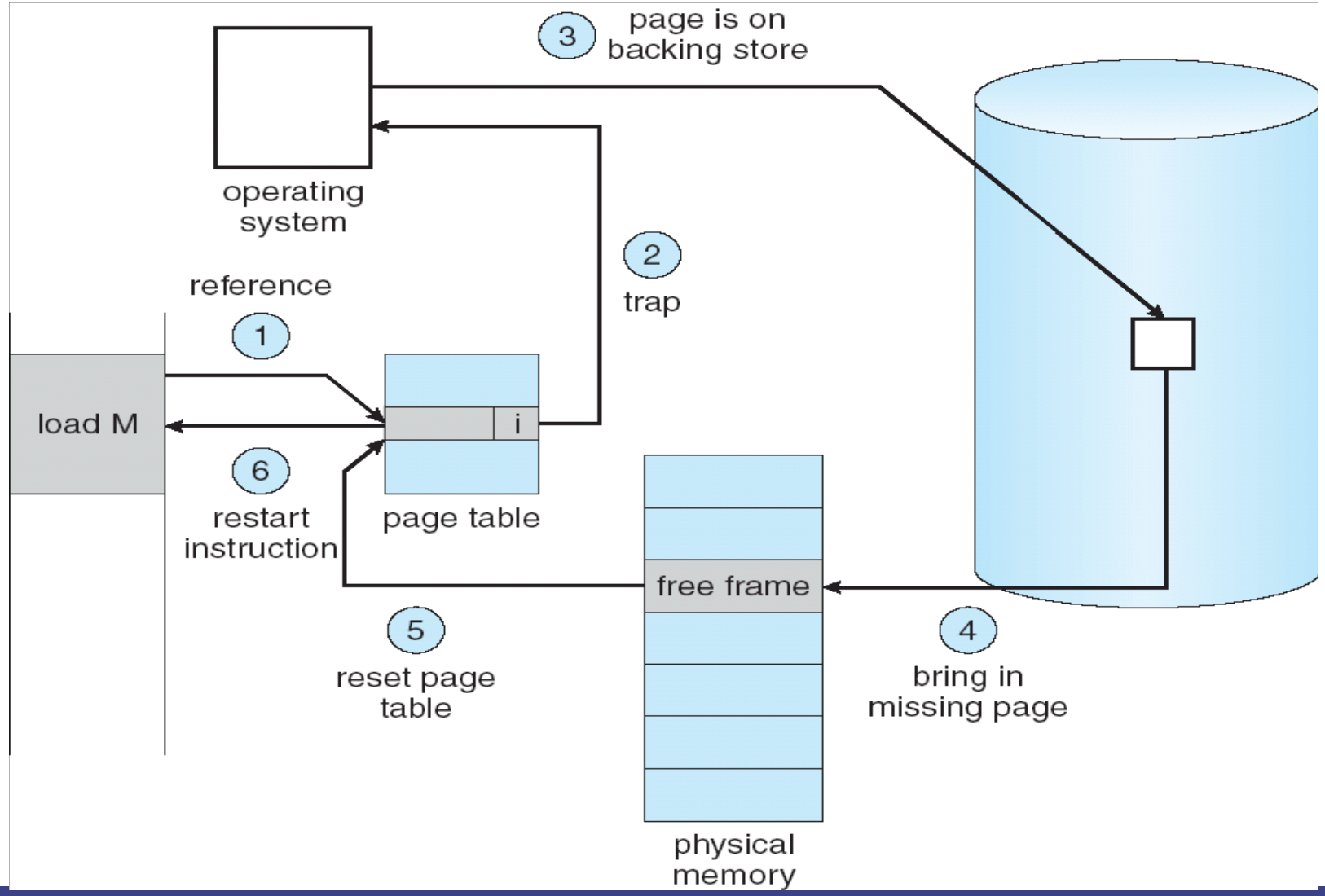
页不在物理内存空间里

3 页错误（page fault）

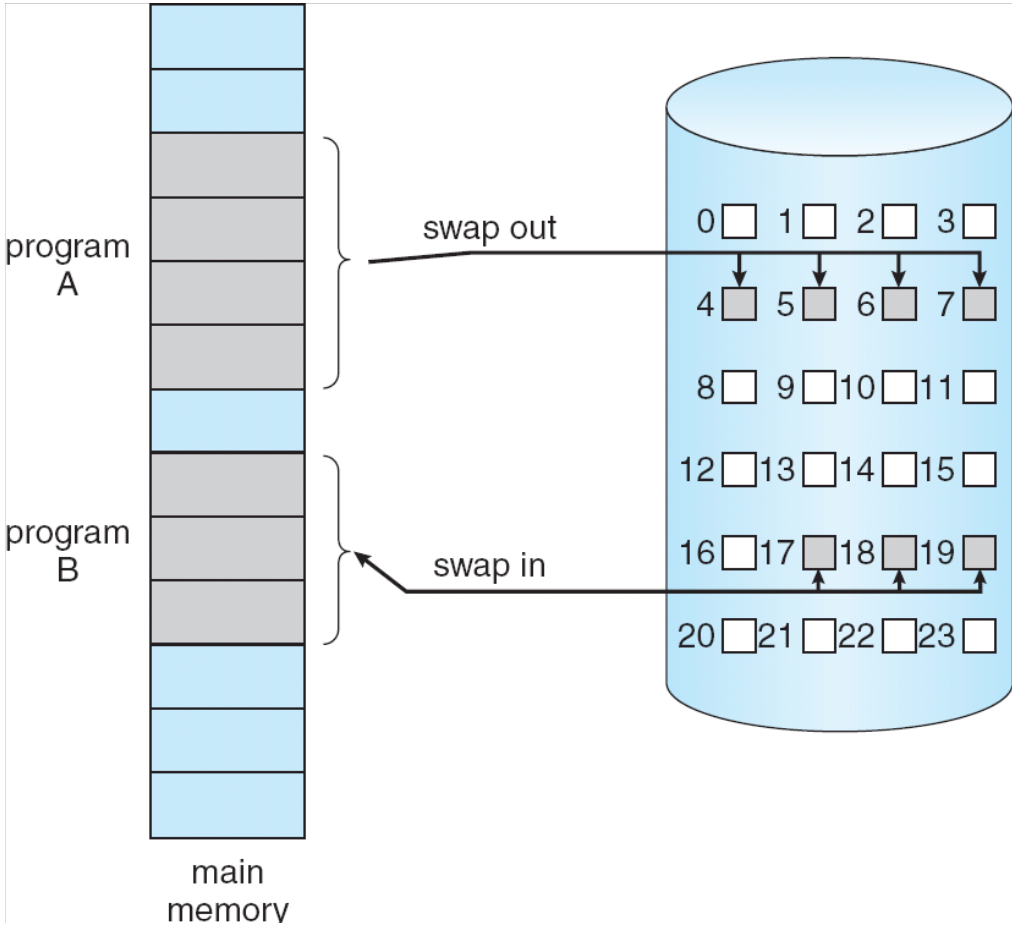
- 想要访问的页不在内存空间（无效页），会发生页错误，并陷入操作系统
- 确定该访问是否合法，不合法就结束进程，合法就进行以下操作（看下一页图）
    I. 找到一个空闲帧
    II. 将所需要的页调入到，找到的空闲帧里
    III. 修改页表（有效、无效位），以表示该页已在物理内存中
    IV. 重新开始因陷阱而中断的指令

问:要是没有空闲帧，怎么办?

处理页错误的步骤


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

分页的内存到连续的磁盘空间之间的传送



- 交换程序
- 调页程序

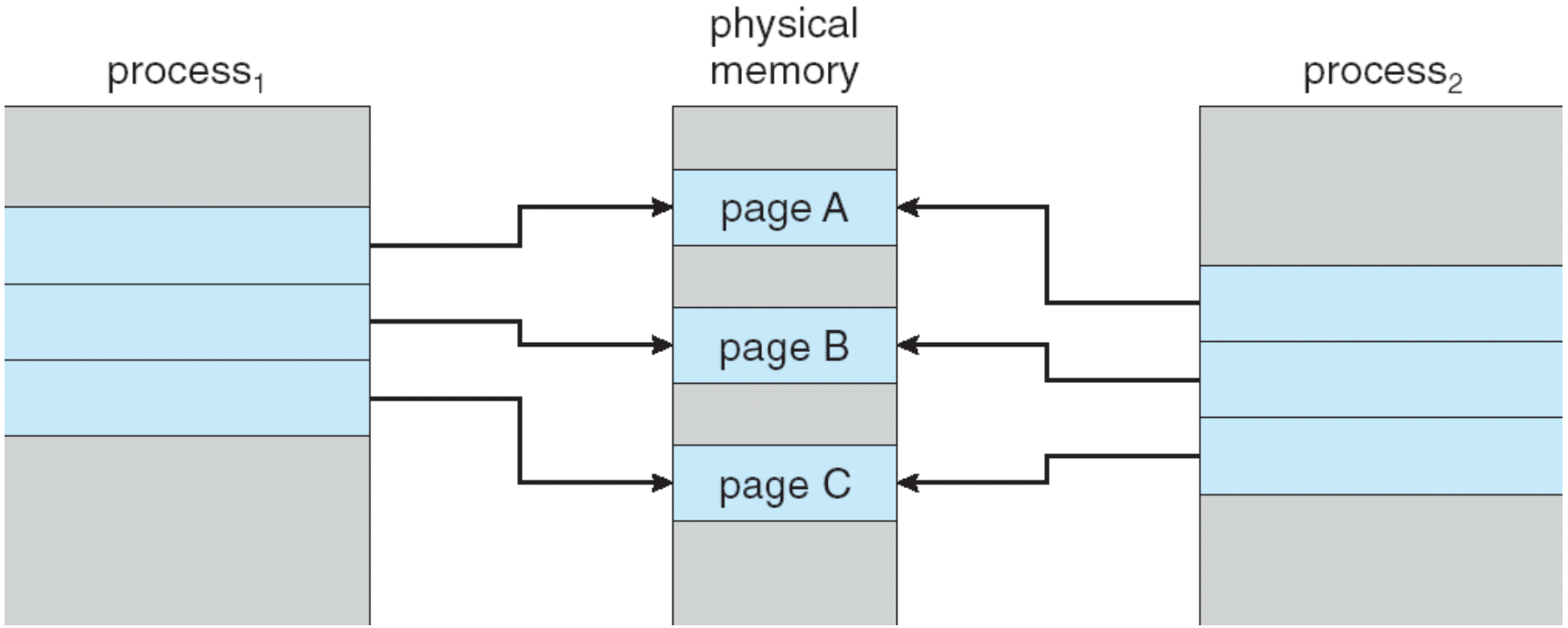
创建进程时使用虚拟内存的长处

- 写时复制(Copy-on-Write)，COW 技术允许父进程和子进程共享物理内存中的页
- 如果任何一个进程需要对页进行写操作，那么就会创建一个共享页的副本
- 优点
    I. 可以快速创建进程
    II. 最小化新创建进程的页数

问:从哪里分配空闲页?答:空闲缓冲池

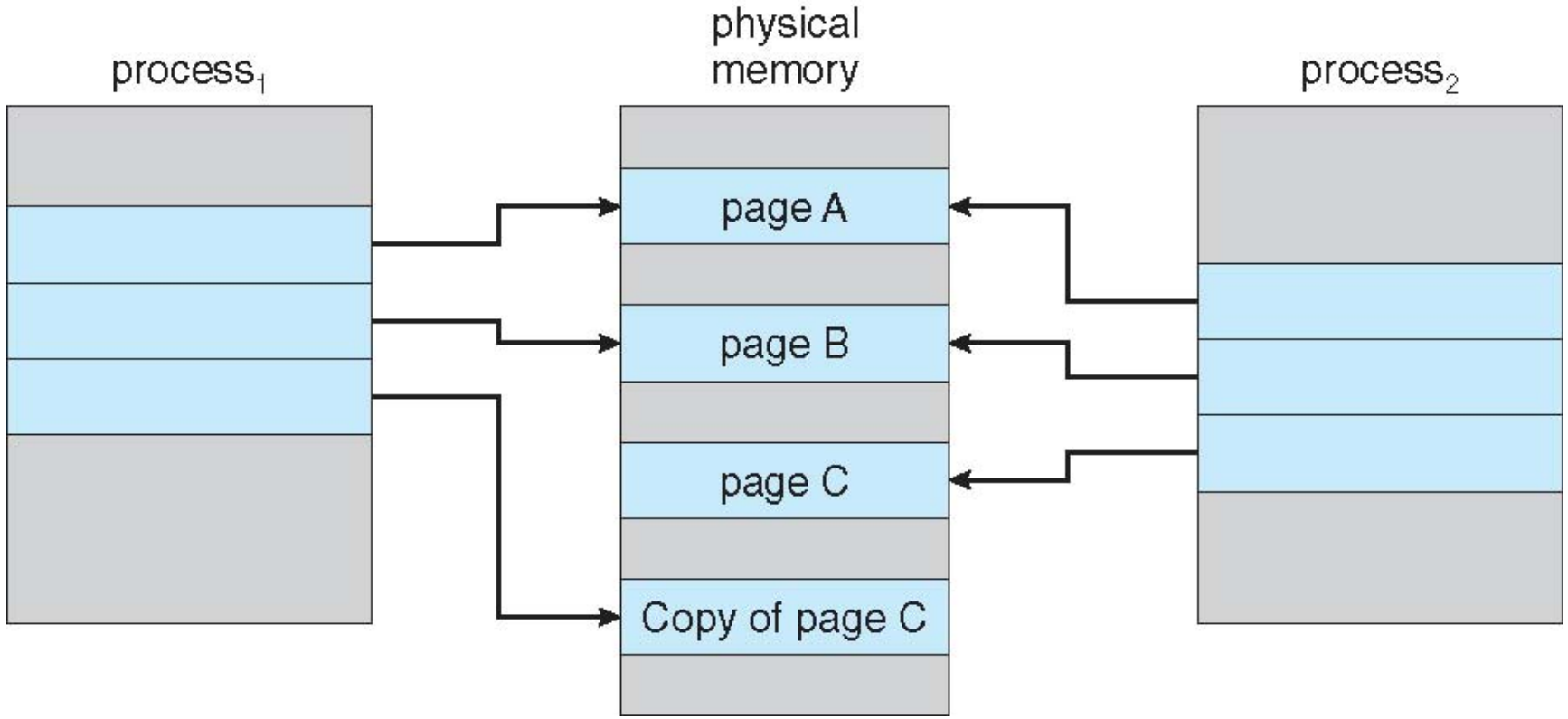
举例:进程1修改页C之前

创建进程时，可以通过共享页的方式，暂不进行复制操作



举例:进程1修改页C之后

当发生写的操作的时候发生复制操作

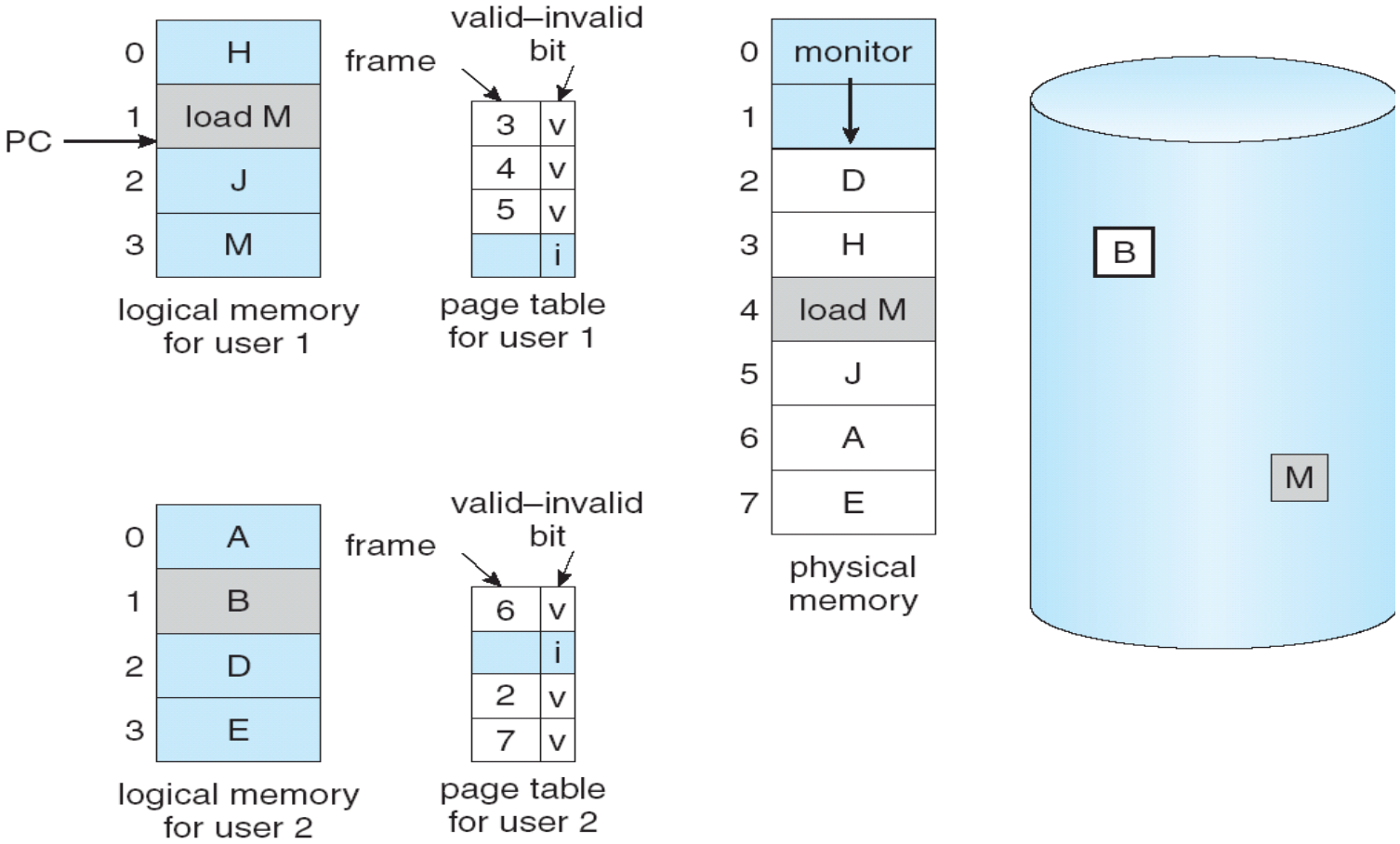


没有空闲帧可分配，怎么办?

- 可以采取以下几种方法
  I. 终止进程
  II. 交换出一个进程，页置换（page replacement）
- 页置换
  I. 找出一个牺牲的帧，并将其换出;
  II. 需要使用的页，将其换入
- 页置换算法的需要考虑的问题
  如何最小化页错误的发生，同一个页有可能多次被释放、被载入

页置换是按需调页的基础，它分开了逻辑内存和物理内存，给程序员提供的巨大的内存空间

Need for page replacement



页面置换的基本操作

A. 查找所需页在磁盘上的位置

B. 查找一个空闲帧

- 如果有空闲帧就用
- 如果没有空闲帧，就通过某种置换算法选择一个“牺牲”帧

C. 将所需要的页读入空闲帧，修改页表和帧表

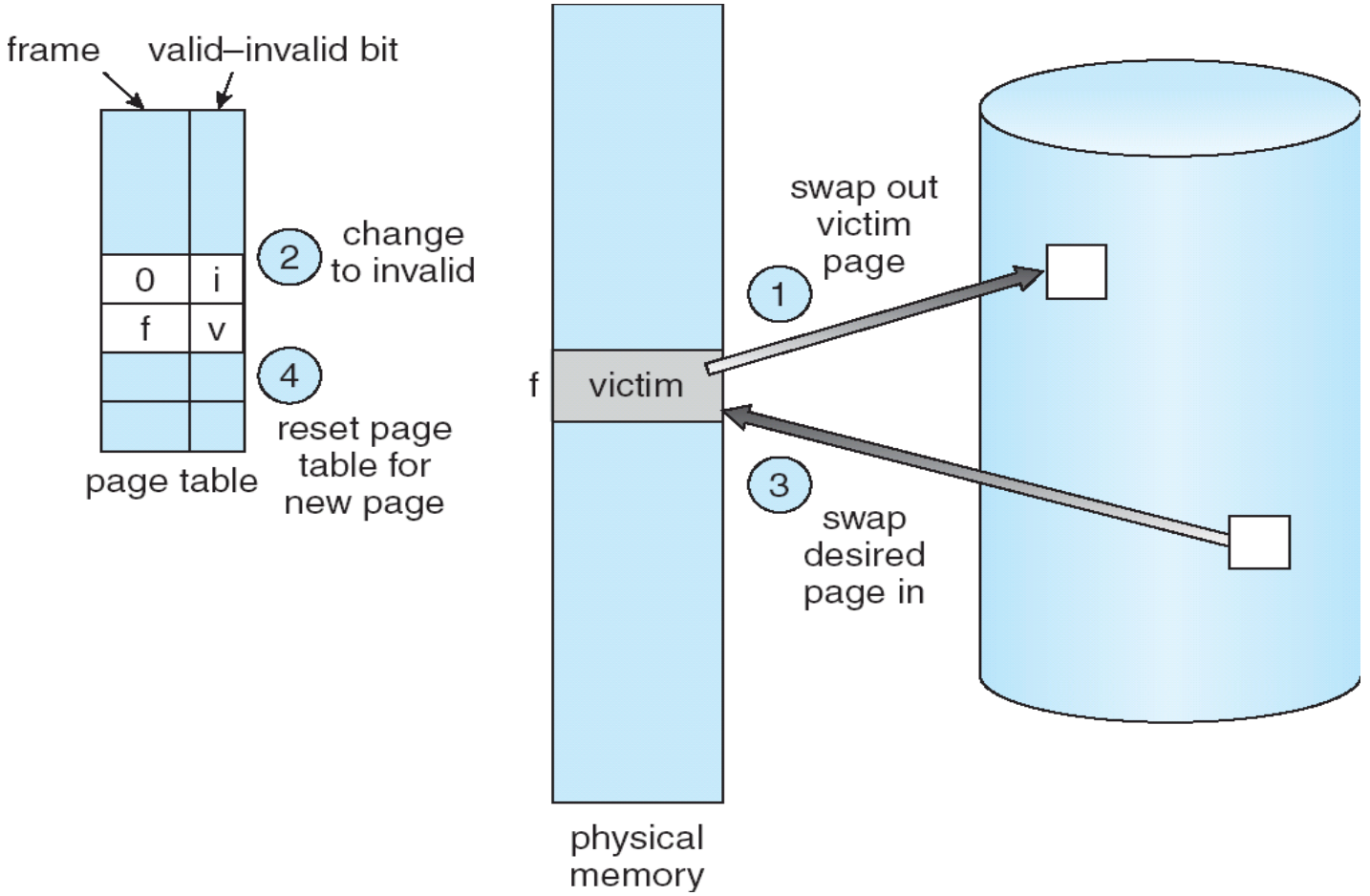
D. 重启进程

页置换

页置换发生两次页传输（换入、换出），导致页处理时间加倍，增加了内存访问时间

有效方法一（换出）

+ 每个页关联一个修改位（modify bit）
+ 通过修改位确认关联页是否被修改
  -- 如被修改过，在换出时必须写入磁盘
  -- 如没有被修改过，在换出时，不需要写入磁盘，从而避免了写入磁盘操作

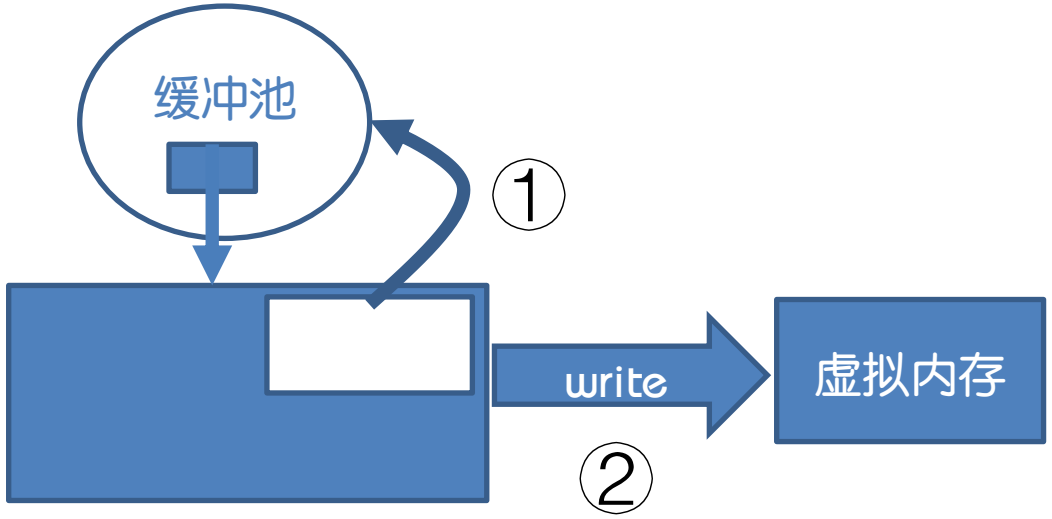

frame
valid–invalid bit
0
i
f
v
page table
2
change
to invalid
4
reset page
table for
new page
f
victim
1
swap out
victim
page
3
swap
desired
page in
physical
memory

有效方法二（换入）:页缓冲

发生页错误，需要页置换，正常应该是先换出牺牲页后，再换入要执行的页

+ 利用空闲帧缓冲池

系统保留一个空闲帧缓冲池，当需要牺牲帧写出虚拟内存时，写出之前，从空闲帧缓冲池中先得到内存（即先分配后换出）



第三节、页置换算法

1. FIFO 算法
2. 最优置换算法
3. LRU (Least Recently Used) 算法
4. 近似 LRU 算法
5. 基于计数的算法

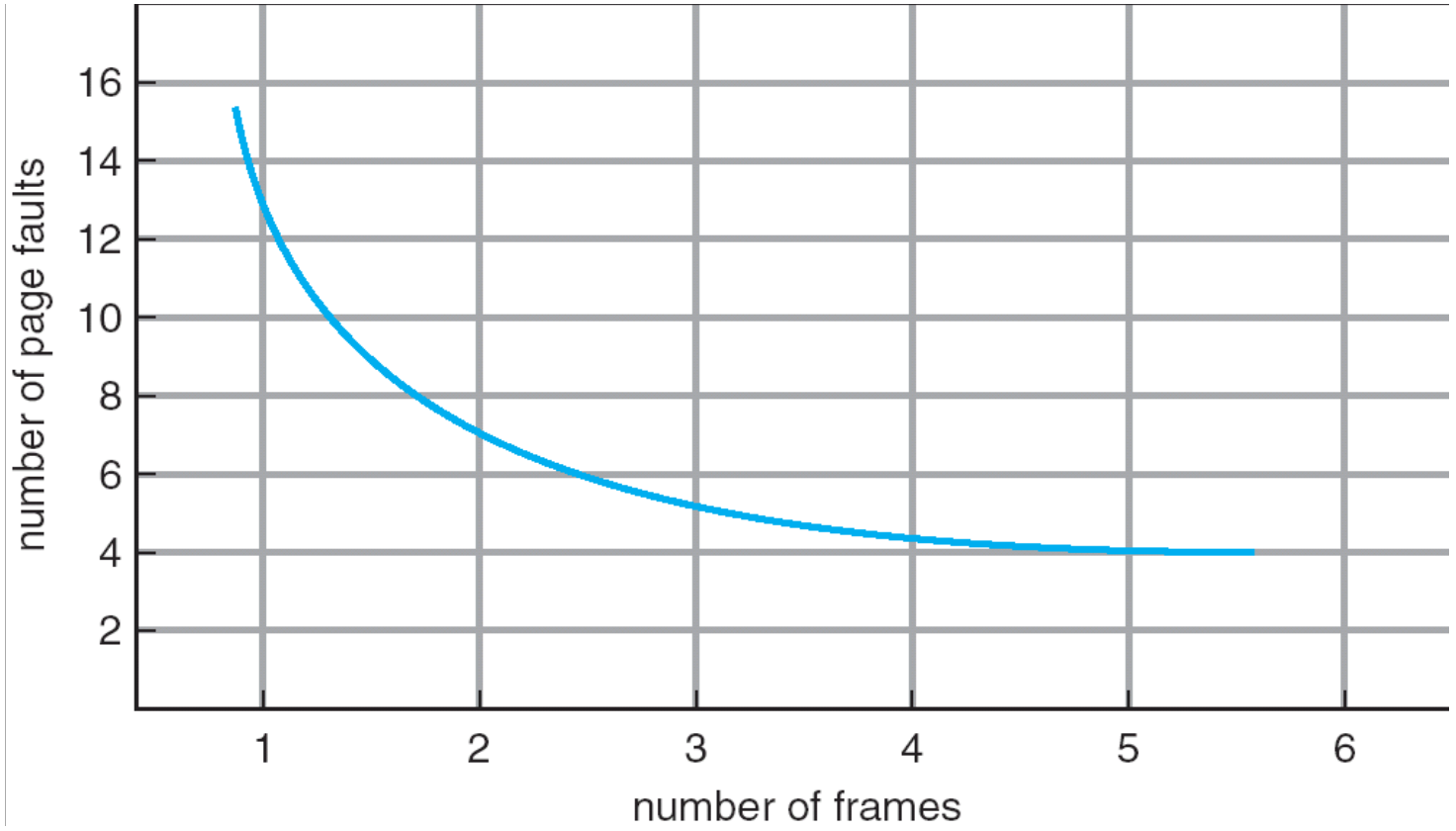
页置换算法的目标是: 最小化页错误的发生

利用引用串来评估一个置换算法

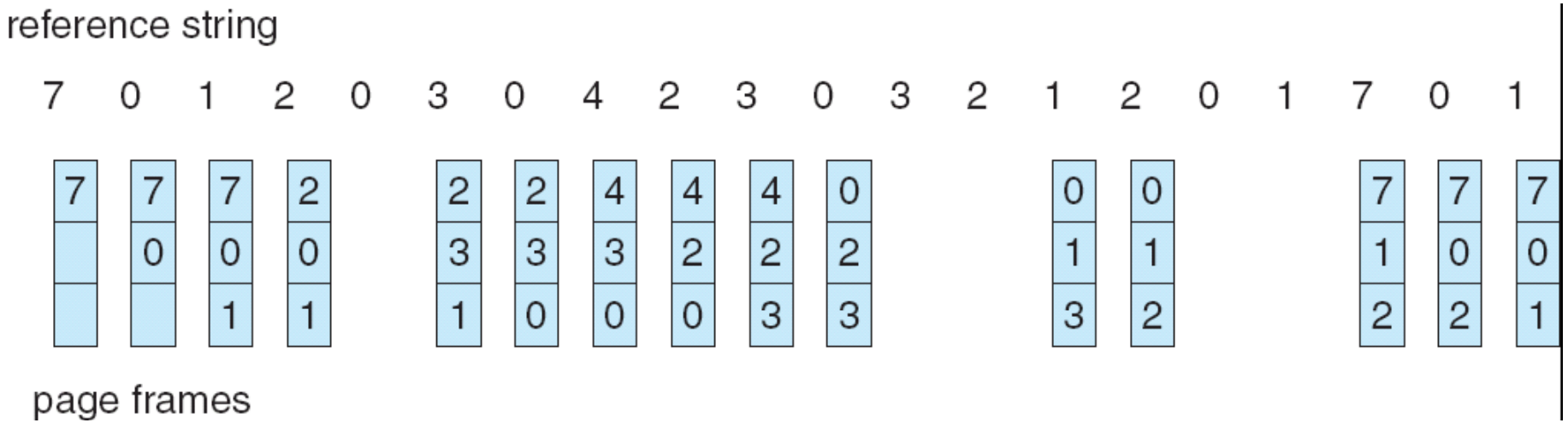
- 引用串: 一系列页的序号
- 评　估: 检查发生的页错误次数

页错误和帧数量关系图

物理帧和页错误成反比
（more frames → less page fault）



(1) FIFO 页置换



- 引用串: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3个帧

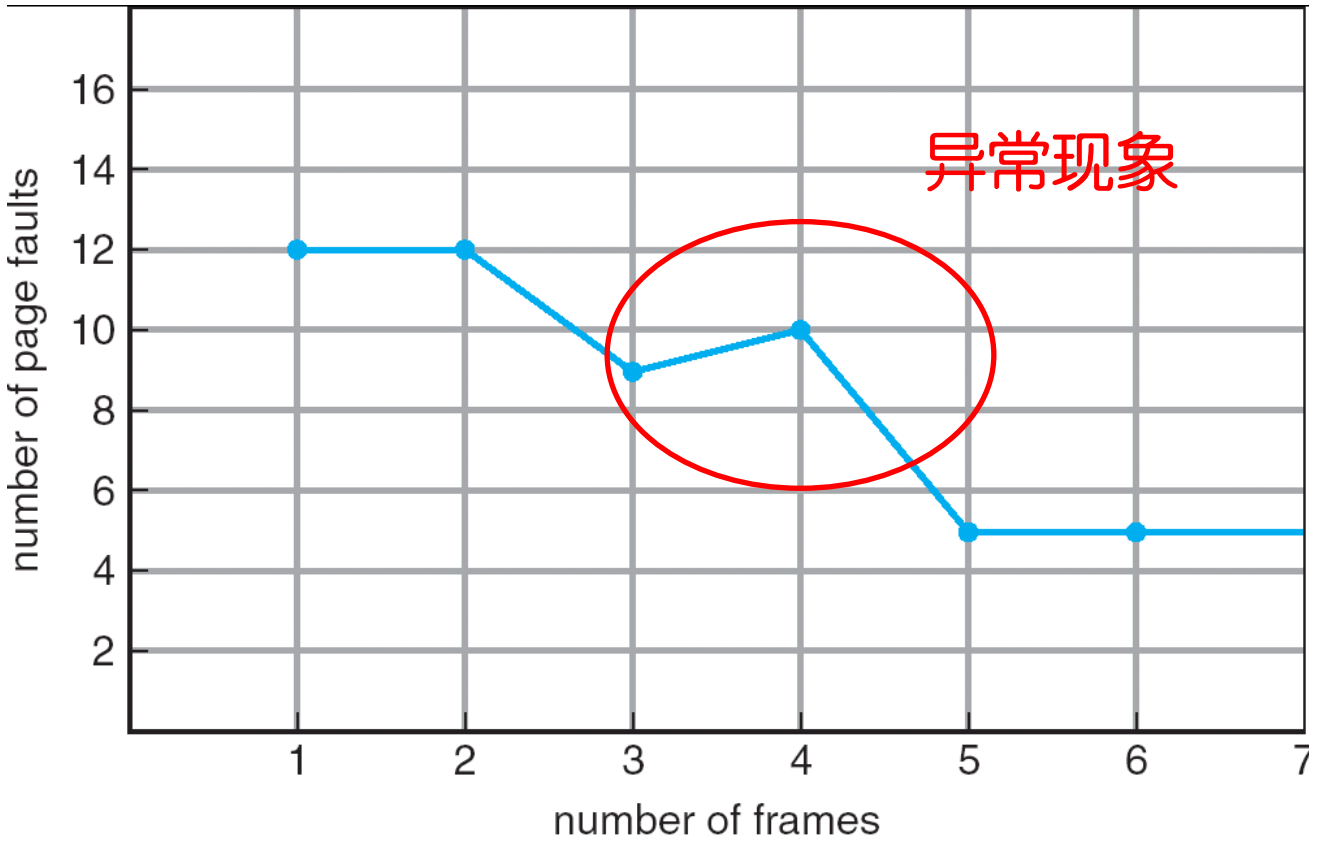
1	4	5
2	1	3
3	2	4

4个帧

1	5	4
2	1	5
3	2	
4	3	

- 异常现象：more frame → more page fault

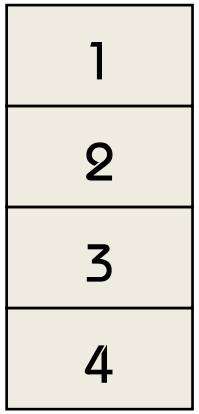
FIFO Illustrating Belady`s Anomaly



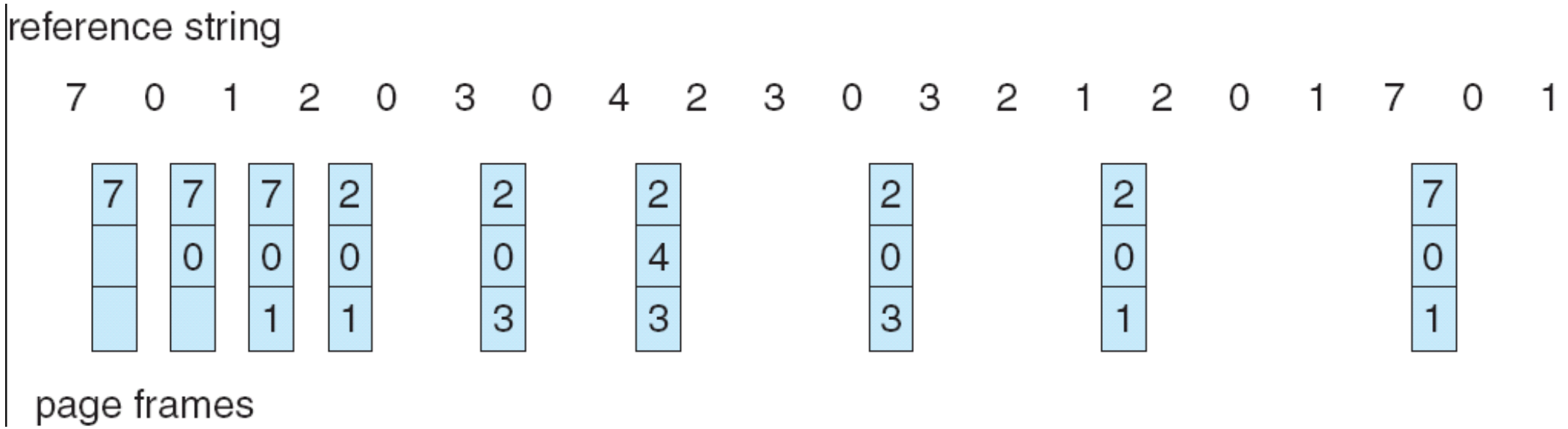
(2) 最优置换

- 置换将来最长时间不会用的帧，4 个帧例子

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- 但问题是怎么知道引用串的未来信息？
- 最优算法主要用于比较研究

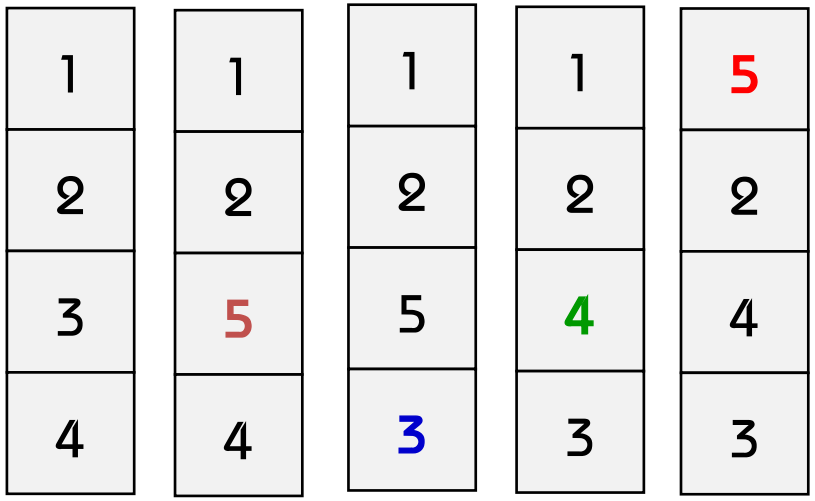


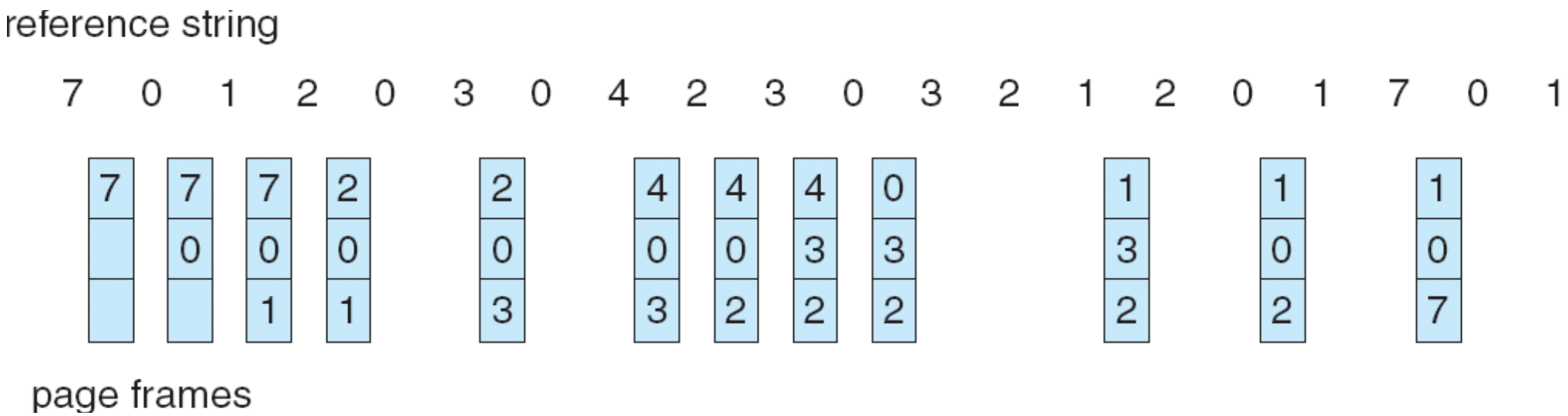
1. 最长时间不会用的是7
2. 最长时间不会用的是1
3. 依次类推
4. 。。

(3) LRU-最近最少使用算法

最近最少使用算法(Least Recently Used):每个页关联该页上次使用的时间，选择最长时间没有使用的帧

- 引用串: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5




reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
page frames

1. 计数器

页表的每一项与计数器相连，并计入时间，但可能会出现如下问题

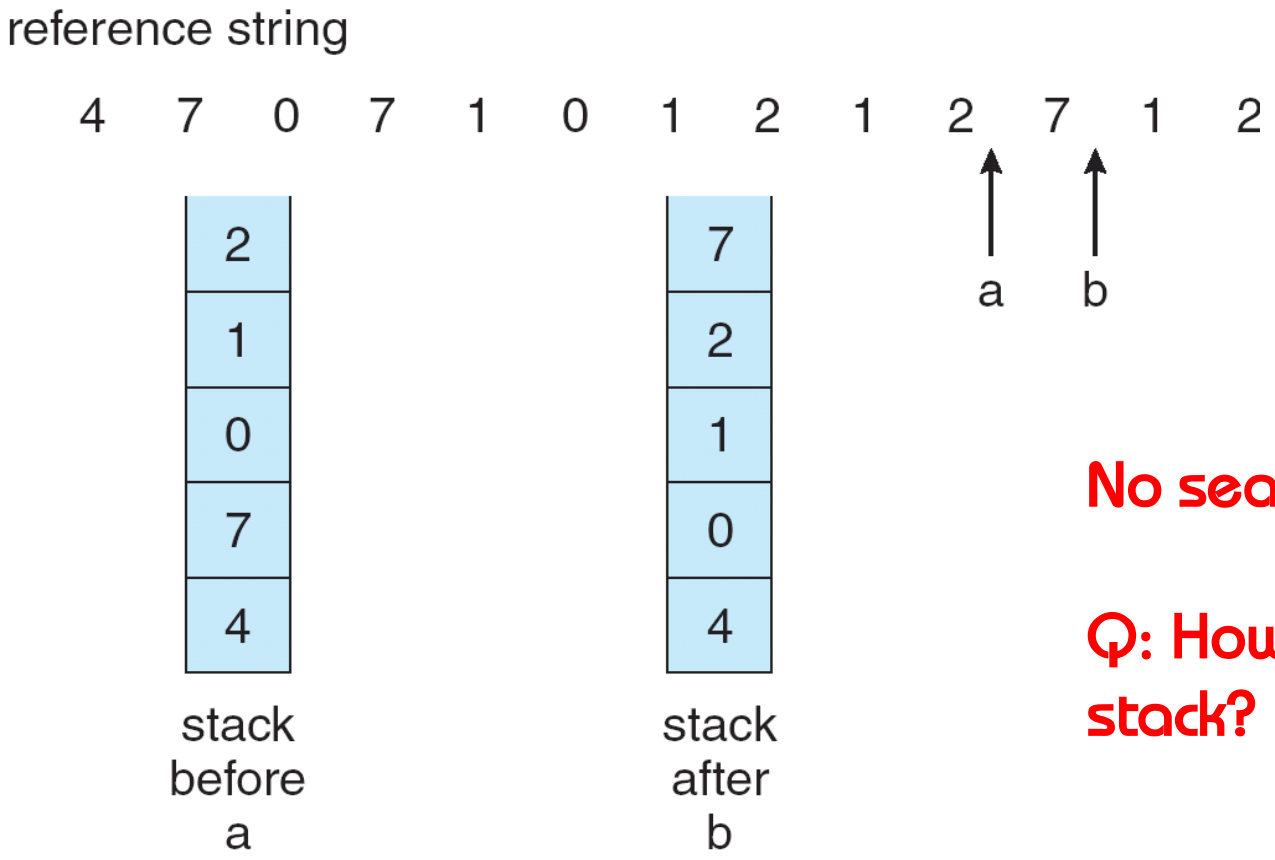
- 增加了访问操作（需记录时间）
- 增加内存使用
- 每次置换需要搜索全部页表

2. 栈

+ 每当引用一个页，该页就移动到栈的最顶部
+ 并依次往下移动
+ 最近不常用的栈放在栈的最低端

采用栈实现方法需要每次更新栈,需要栈中项的移动

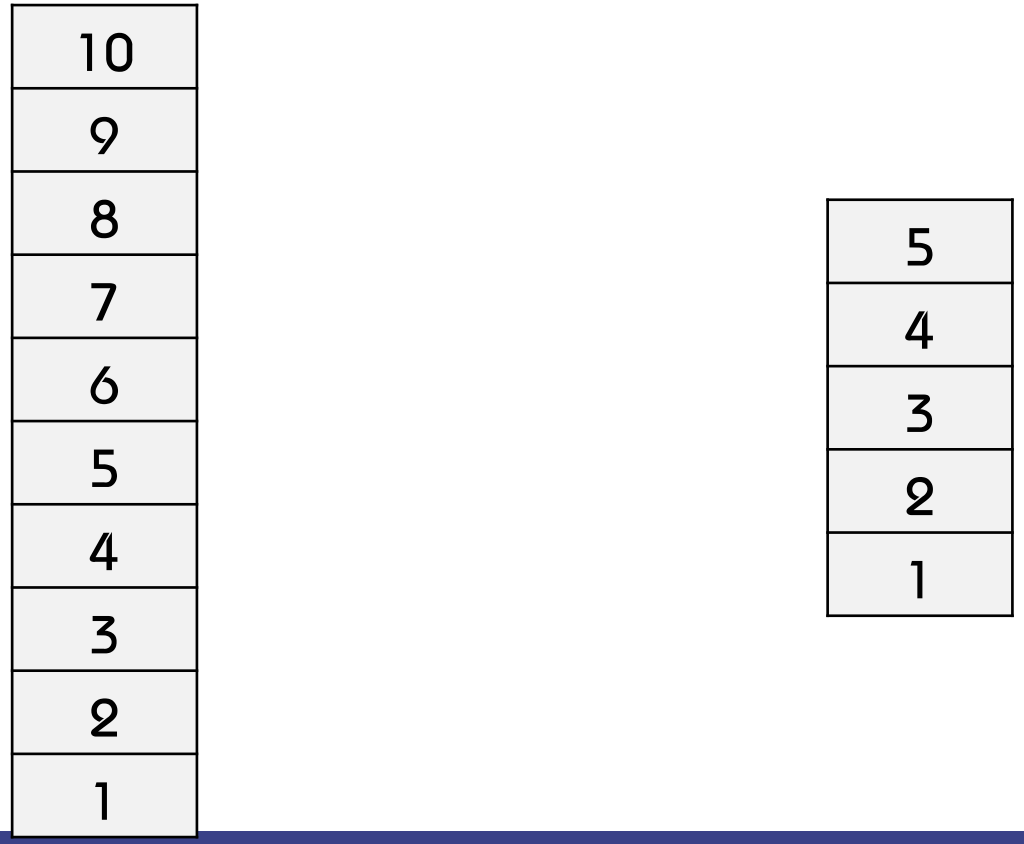
用栈来记录最近使用的页



No search for replacement

Q: How to determine the depth of the stack?

- We assume that a process will reference 10 number of pages,
- Let us consider the depth of stack is 10 and 5
- Reference string 1 2 3 4 5 6 7 8 9 10, 1 2 3 4 5 6 7 8 9 10,



(4) 近似LRU算法

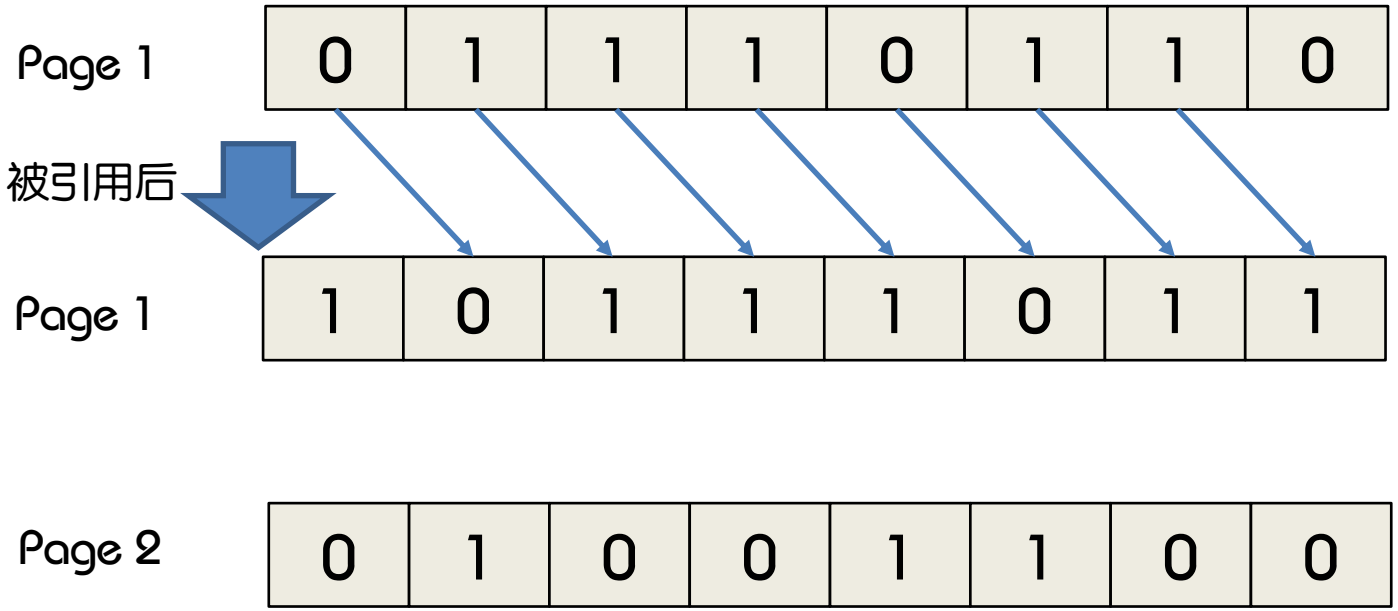
1. 附加引用位算法
2. 二次机会算法
3. 增强型二次机会算法

1. 附加引用位算法

- 每个页都与引用位相关联
- 每当引用页时，相应页的引用位就被硬件置位
- 开始，引用位被初始化为0
- 页被引用，引用位被设置为1；没被引用，就设置为0（在规定的时间周期）
- 如 8个字节的引用位表示对8个周期进行记录引用位
- 每次引用的记录，放到8位字节的最高位，而将其他位向右移一位，并抛弃最低位

0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1

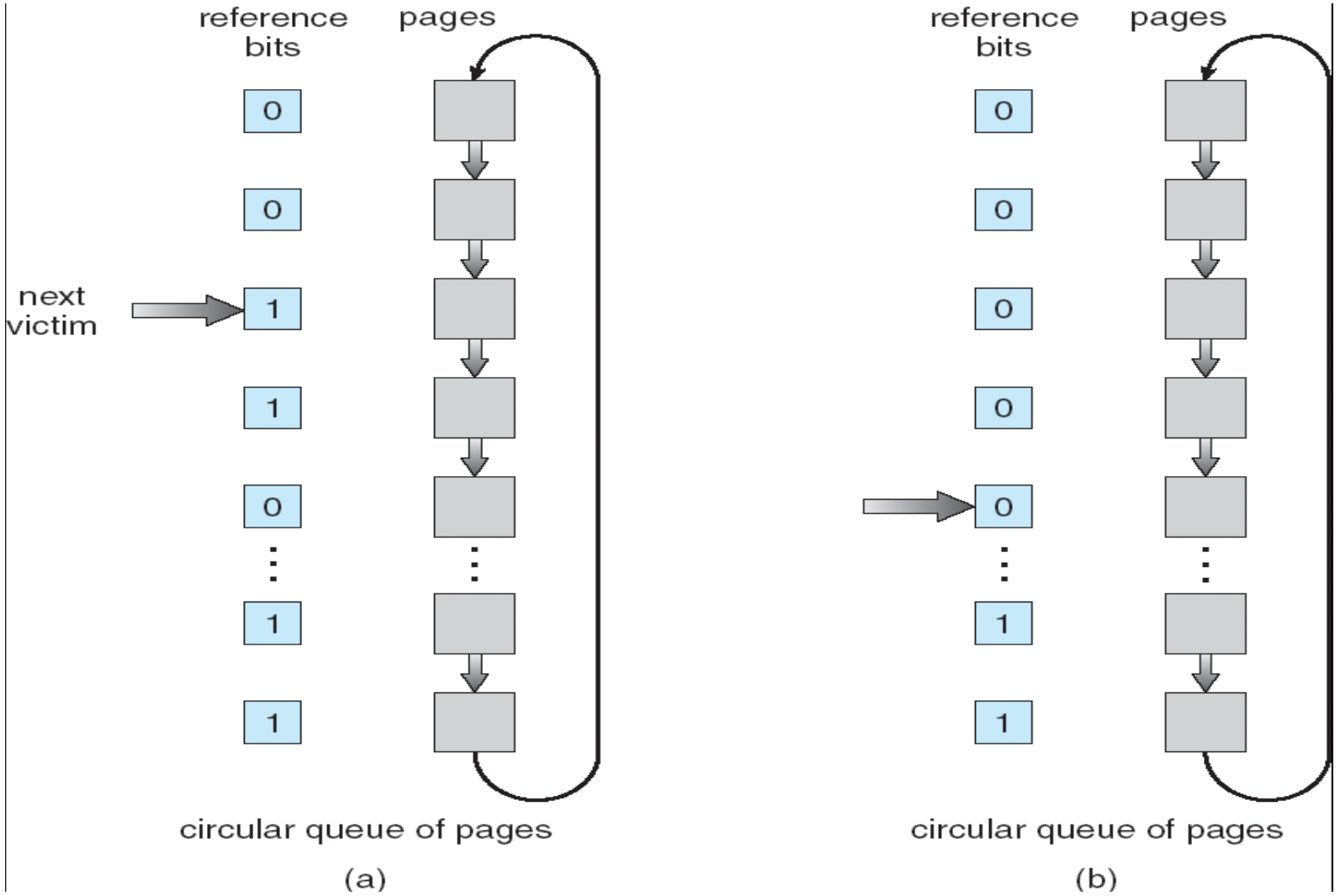


如举例，每个页都有自己的引用位的值，哪个引用位的值最小，就替换哪个

每当需要置换页时，检查每页相关联的引用位，

1. 如引用位为 0 就置换，并把引用位设置成 1
2. 如引用位为 1 就跳过（给第二次机会），并清零，然后跳到下一个FIFO 页

II. 二次机会算法


reference bits
pages
0
0
next victim
1
1
0
1
1
circular queue of pages
(a)
reference bits
pages
0
0
0
0
0
1
1
circular queue of pages
(b)

III. 增强型二次机会算法

- 利用二个位，即引用位和修改位
  + 第一位表示是否被引用过
  + 第二位表示是否被修改过

- 采用这两个位，有以下可能类型
  + （0, 0）最近没有使用且也没有修改，用于置换最佳页
  + （0, 1）最近没有使用但修改过，需要写出到磁盘
  + （1, 0）最近使用过但没有修改，有可能很快又要被使用
  + （1, 1）最近使用过且修改过，有可能很快又要被使用，置换时需要写出到磁盘。

(4) 基于计数的页置换

保留一个用于记录其引用次数的计数器

I. 最不经常使用页置换算法

+ 理由:经常活动的页应该有更大的引用次数

II. 最常使用页置换算法

+ 理由:引用次数少的页可能是刚刚调进来的，但将来可能经常用

第四节、物理帧的分配

6. 物理帧的分配

- 每个进程需要分配最小需要运行的页
- 帧的分配有如下方式:

平均分配方式

:每个进程分配物理帧的大小相同

比例分配方式

:根据进程大小比例

优先级分配方式

:根据进程优先级分配

页置换分类:全局分配和局部分配

页置换可以分为全局置换和局部置换两大类

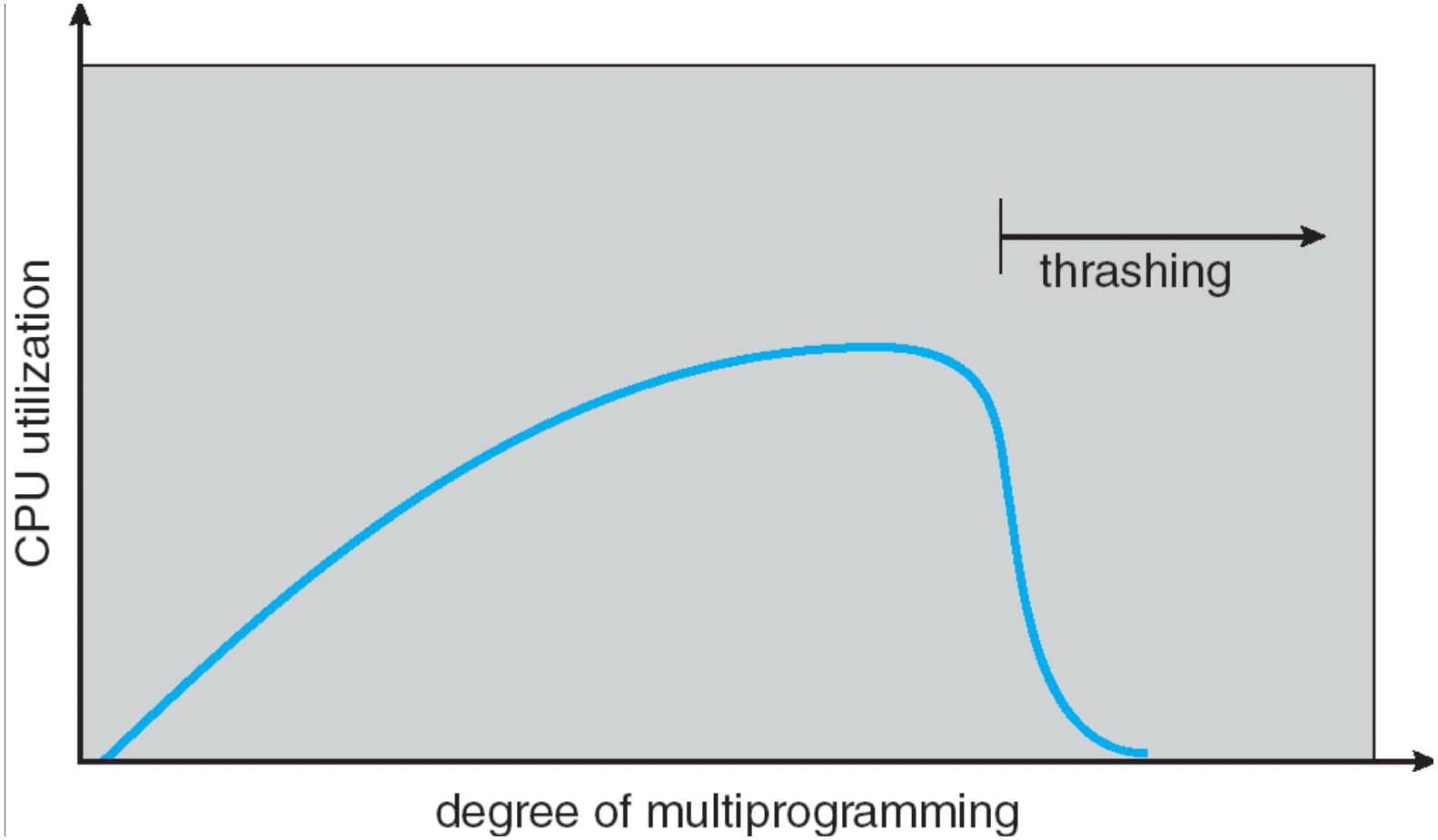
1. **全局置换**:从所有帧集中选择一个置换帧
   - 分配到的帧数量可能发生变化
2. **局部置换**:仅从自己的分配帧中选择一个置换帧
   - 分配到的帧数量不会发生变化

系统颠簸（Thrashing）

因一个进程没有分配到“足够”的页帧，而频繁的发生页错误，这会导致:

1. CPU 使用率下降，操作系统会试图增加多道程序的程度
2. 进程会试图去强别的进程的帧

Thrashing ≡ a process is busy swapping pages in and out


CPU utilization
thrashing
degree of multiprogramming

7. 内存映射文件

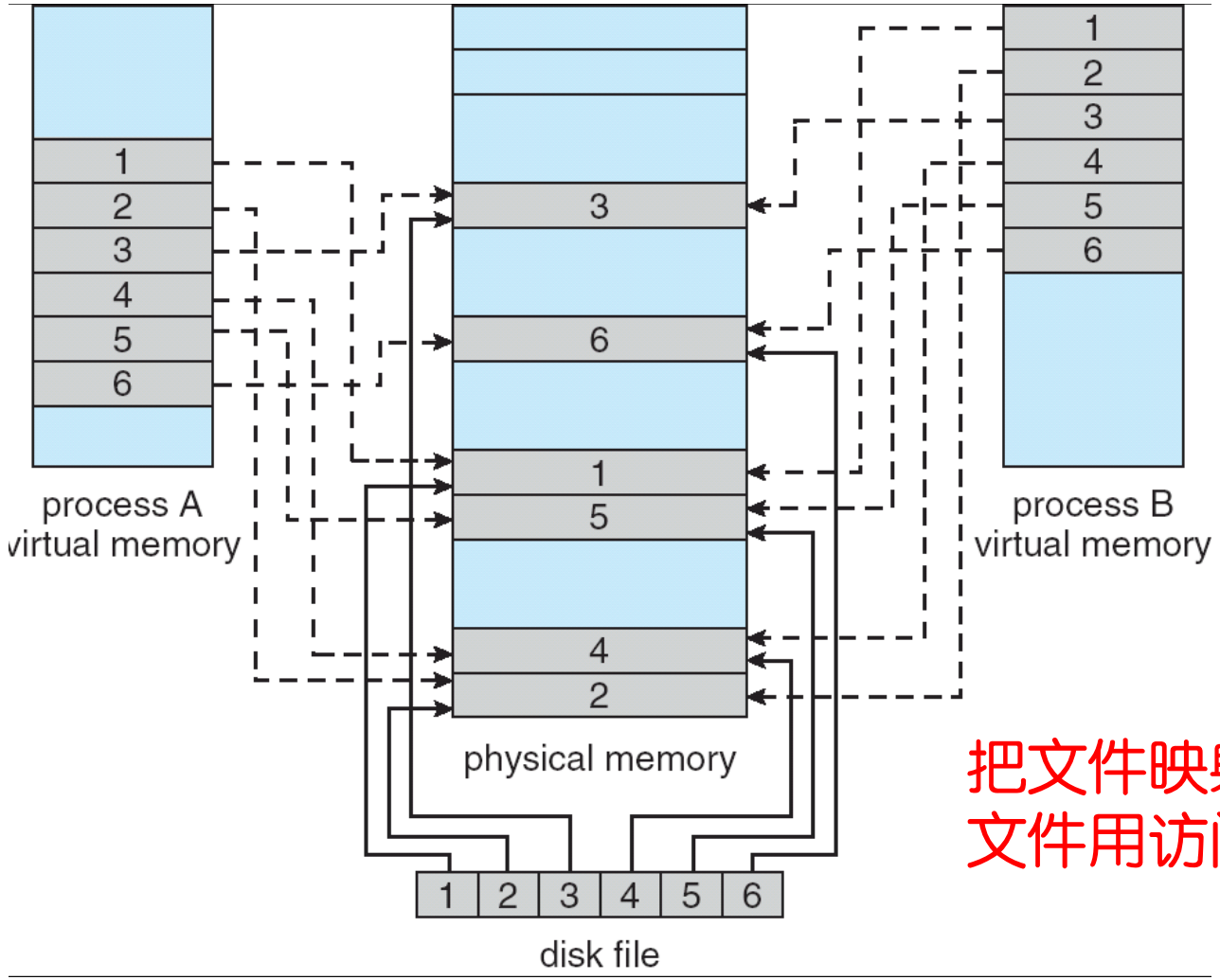
- 如何对磁盘上的文件进行一系列的操作，实现象 open(), read(), write() 等系统调用，可采用以下两种方式

1. 对磁盘上的文件进行直接的操作
2. 利用虚拟内存技术→ 内存映射文件

内存映射文件

- 将磁盘块儿（block）映射成内存的一页或多页
  + 访问文件会发生页错误
  + 文件的读写就按通常的内存访问来处理
  + 文件的写（磁盘）I/O 操作不会立即发生，而是定期的发生或关闭文件时发生
- 优点- 文件共享

多个进程可以将同一个文件映射到各自的虚拟内存中，以允许数据共享



把文件映射到内存，访问文件用访问内存的方式

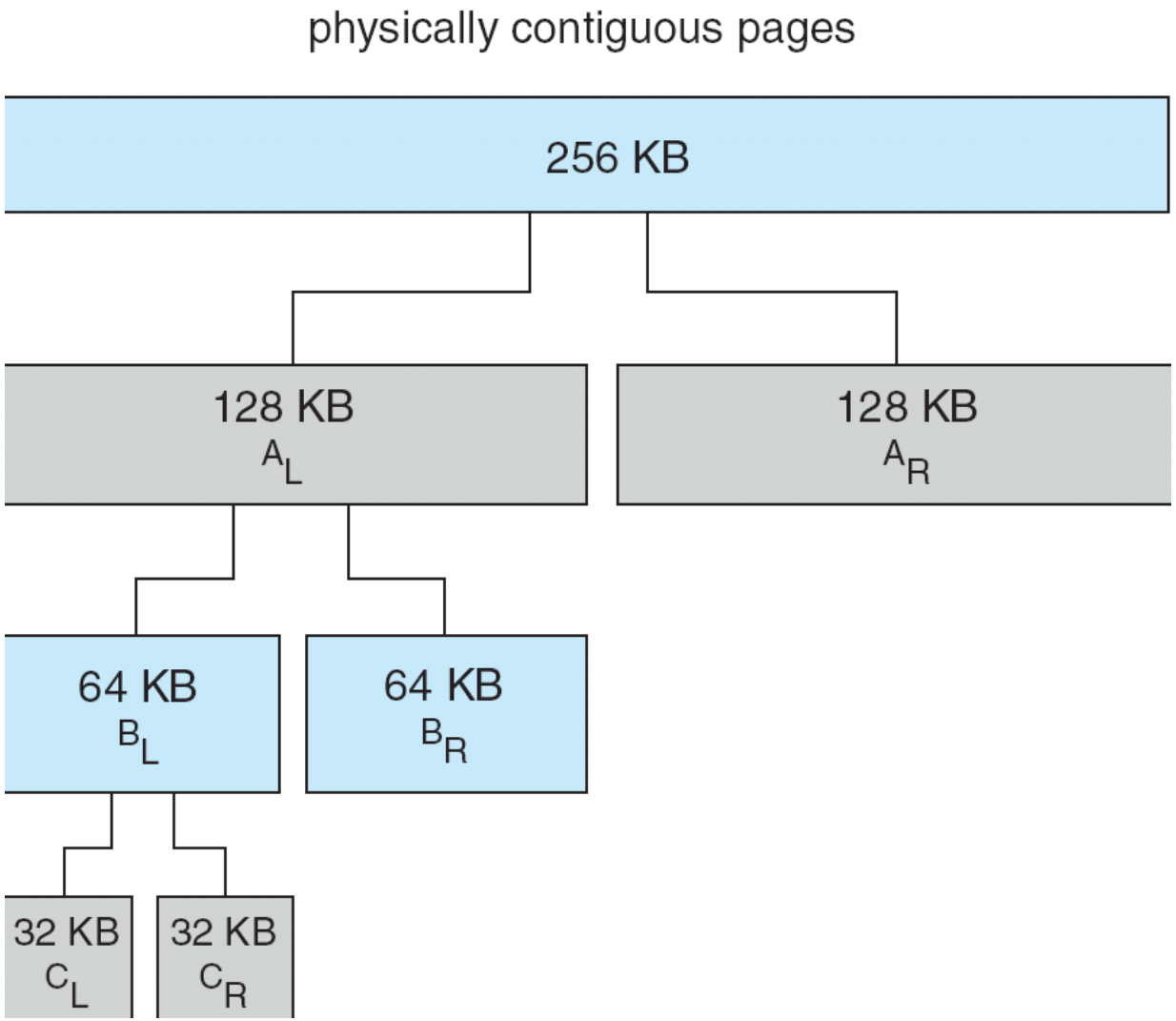
- 与用户内存的分配方式不同，不受分页系统的控制

- 通常从空闲内存池中获取，内核
  + 需要为不同大小的（内核）数据结构，
  + 需要连续分配
    (1) buddy 分配
    (2) slab 分配

(1) buddy 分配

- 从物理上连续的、大小固定的段上进行分配
- 内存按2的幂（power of 2）的大小来进行分配（2,4,8,16...），直到分配合适的页为止

- 如需要12k，分配16k.

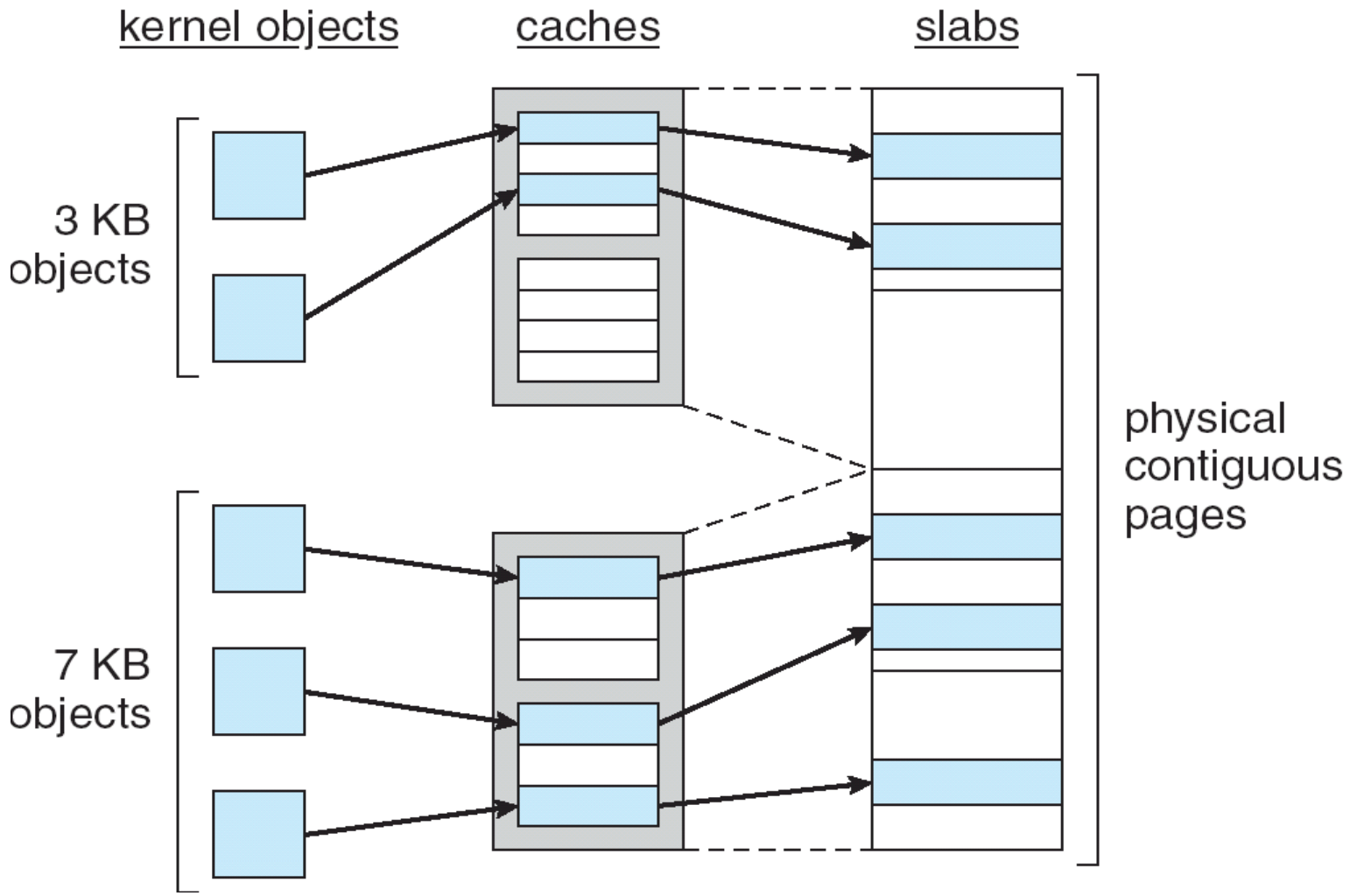

physically contiguous pages
256 KB
128 KB
A_L
128 KB
A_R
64 KB
B_L
64 KB
B_R
32 KB
C_L
32 KB
C_R

(2) slab 分配

1. slab 是由一个或多个物理上连续的页
2. cache 含有一个或多个 slab

3. 每个内核数据结构（信号量，文件对象，进程描述符等）都有它的cache，每个cache 含有内核数据结构的对象实例
4. 当创建cache 时，起初包含若干标记为空闲的对象，对象的数量与 slab 的大小关联
5. 当需要内核数据结构的对象时，可从cache中获取，并标记为使用

slab 分配


kernel objects
caches
slabs
3 KB objects
7 KB objects
physical contiguous pages

Q&A