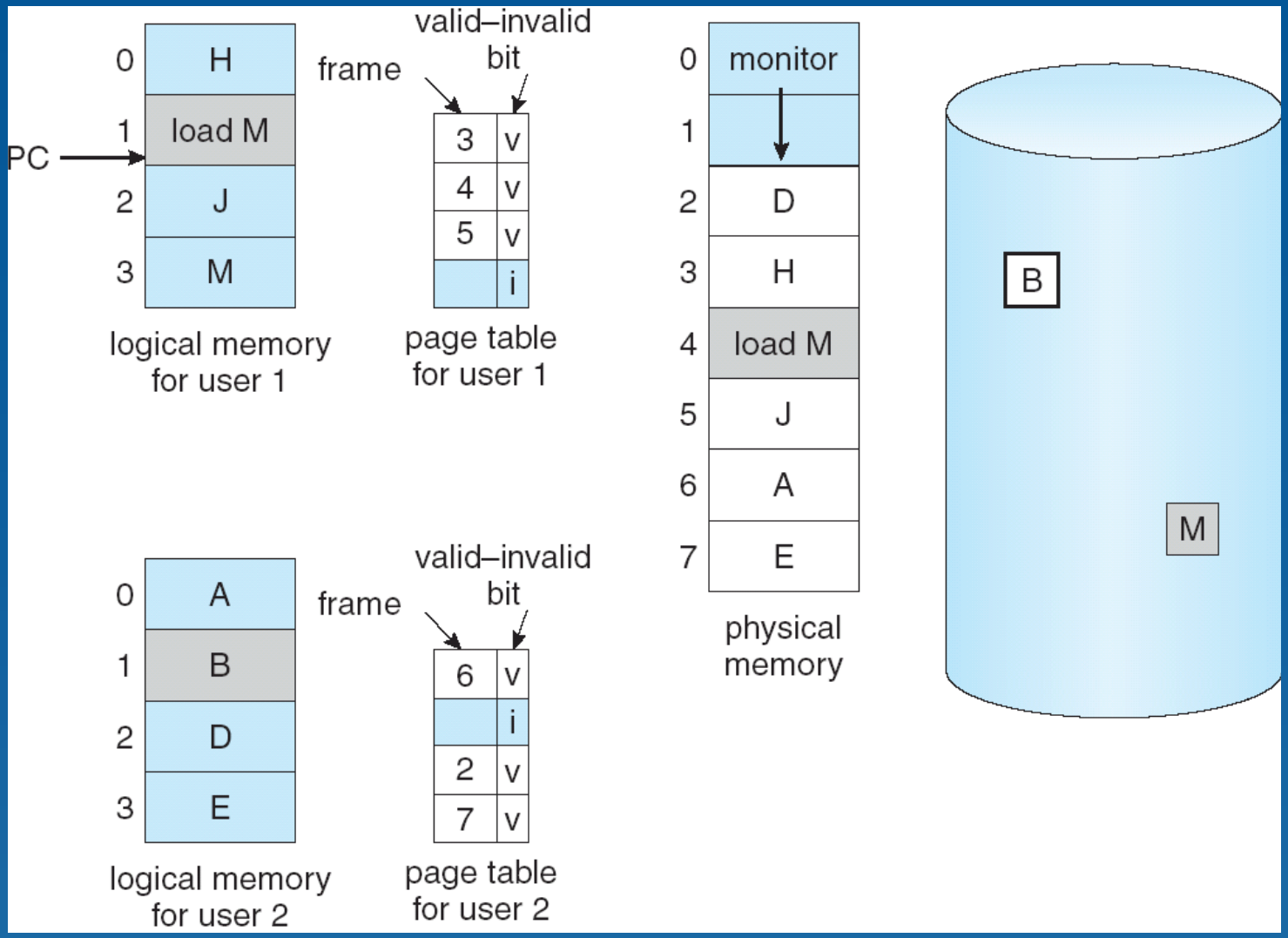
页面置换 (Page Replacement)

缺页响应时，如果没有空闲页帧 ？



页面置换 (Page Replacement)

- 页面置换 – 在内存中找出某个逻辑页面，把它换出。需考虑
- 页面置换算法
- 性能的影响 – 选中的页面置换算法，使之引起的缺页中断次数最少

- 由于程序执行的不可知性，同一页面可能会装入多次

按需调页策略的性能分析

- 设缺页率 (Page Fault Rate)， $0 \le p \le 1.0$
  - if $p = 0$，没有缺页
  - if $p = 1$，每次页面引用总引起缺页
- 有效访问时间， Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{内存访问时间} + p \times (\text{缺页中断处理开销} + \text{页面换出时间} + \text{页面换入时间} + \text{重新执行指令的开销})$$

示例：按需调页策略的性能

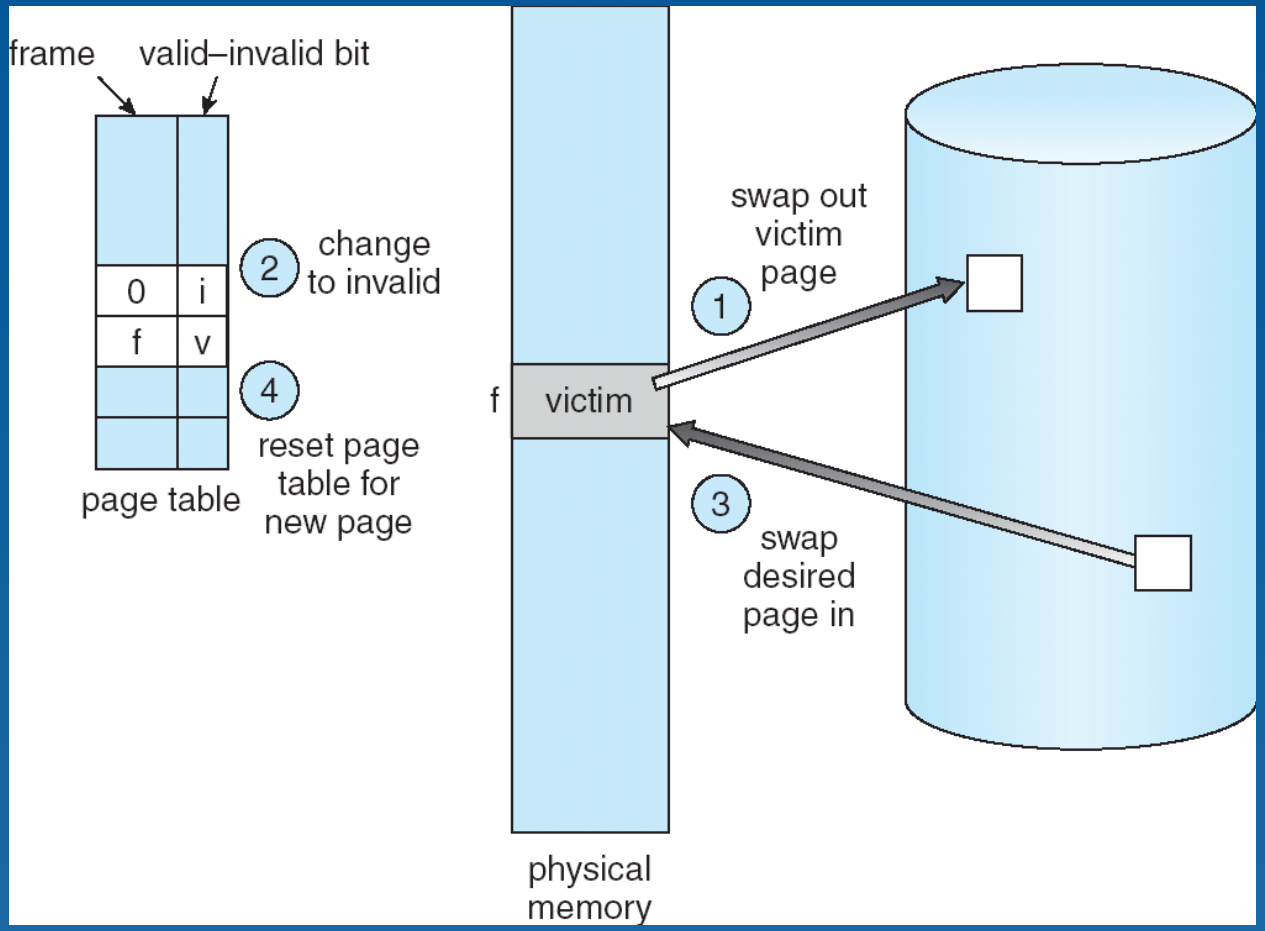
- 设，内存访问时间 = 200 纳秒
- 设，缺页处理平均时间 = 8 毫秒
- EAT = (1 – p) x 200 + p (8 毫秒)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- 假设 1000 次页面引用会引起一次缺页中断，则

  EAT = 8.2 微秒
- EAT 比正常的内存访问慢约 40 倍 !!

页面置换流程


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

页面置换基本流程

1. 缺页中断响应，先确定逻辑页面在交换区的位置
2. 需要一个空闲页帧：

- 如果找到空闲页帧，就选它了
- 如果找不到空闲页帧，调用页面置换算法选择一个“牺牲者”页帧，换出位于该页帧的页面

3. 将目标页面换入至（刚腾空的）空闲页帧。更新页表等数据结构
4. 返回到引起中断的进程。重新执行引起中断的指令

页面置换中一个技巧

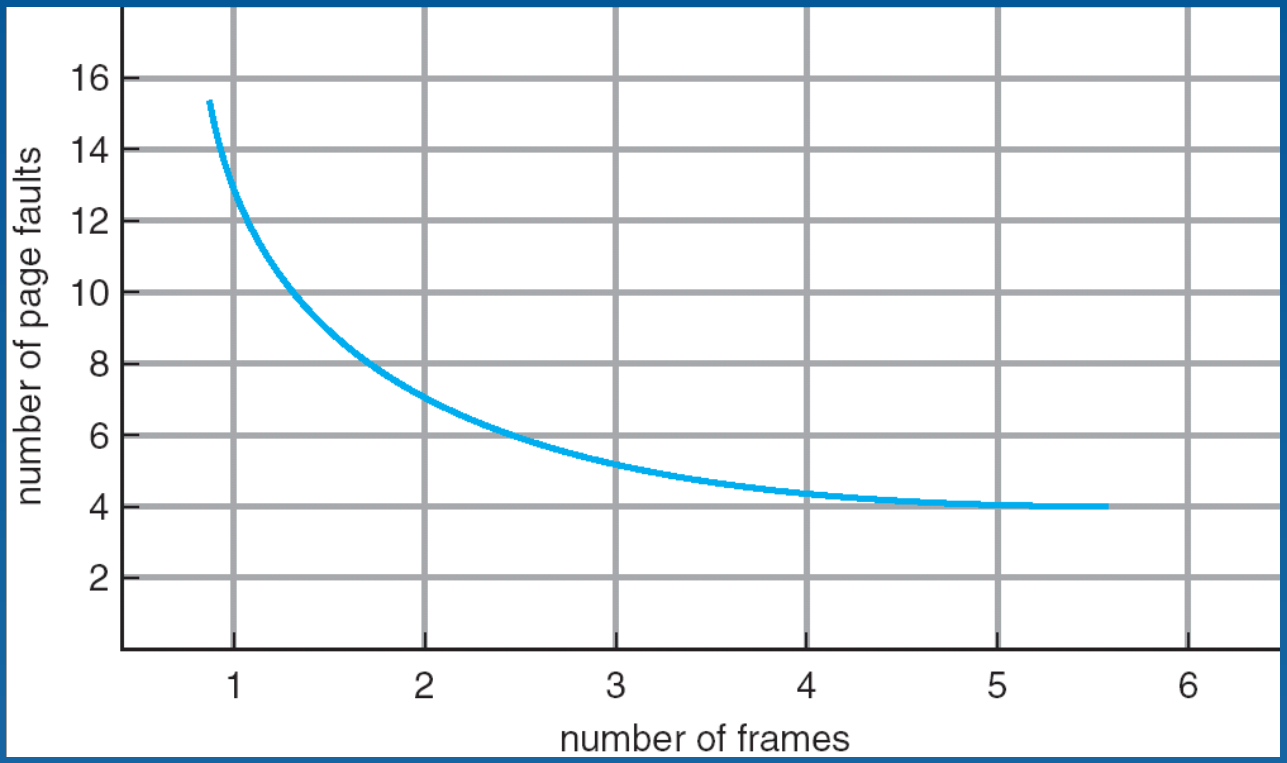
- 设计利用 **modify (dirty) bit** 减少页面传输的开销
- 换出页面时判断：该页面装入后，没被修改过，就不需换出，可直接丢弃。因为交换区里有它的备份
- 只有被修改过的页面，才有必要换出，也就是更新交换区的备份

分析页面置换算法

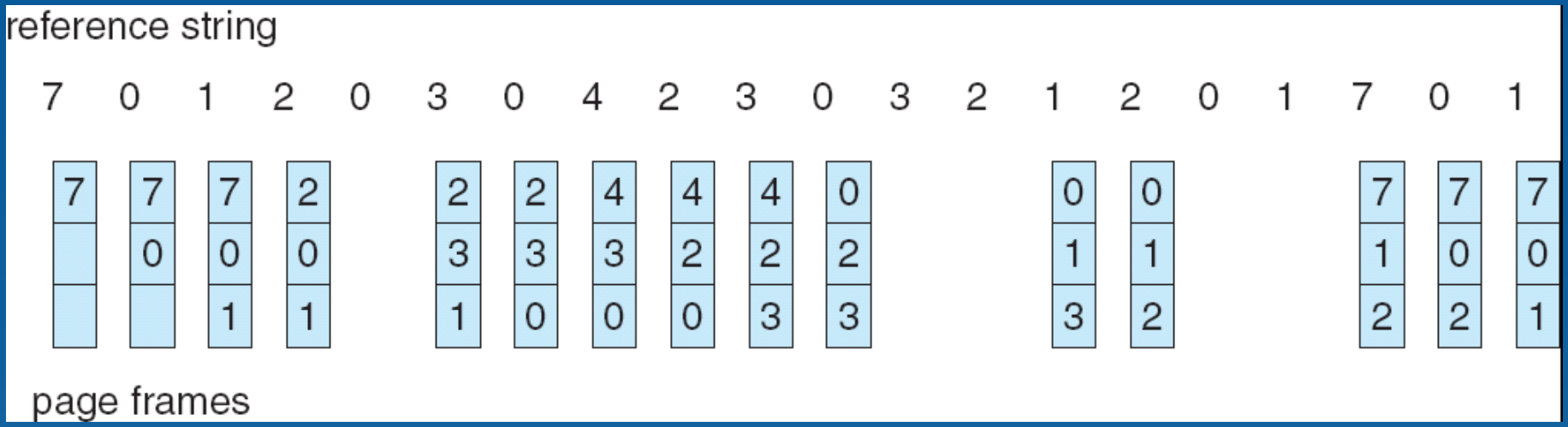
- 追求最低的缺页率
- 评估方法：让算法响应一串特定的内存引用（引用串，reference string），累计其缺页次数
- 后续算法讨论中，统一使用引用串

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

缺页次数相对于物理页帧数的关系



First-In-First-Out (FIFO) 算法

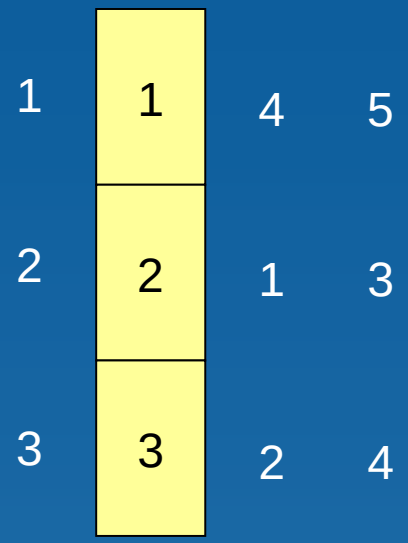


First-In-First-Out (FIFO) 算法

- 引用串

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

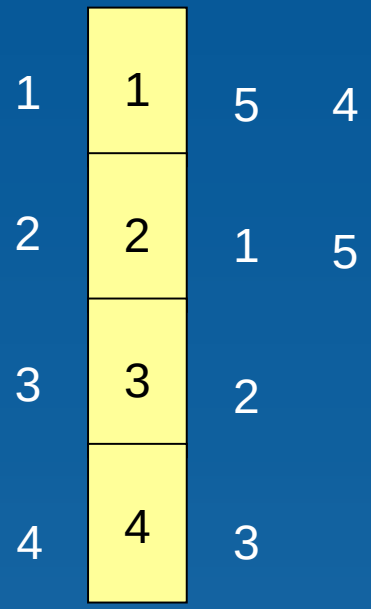
- 3 个物理页帧（每个进程用 3 个页面）



缺页 9 次

First-In-First-Out (FIFO) 算法

- 4 个物理页帧（1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5）

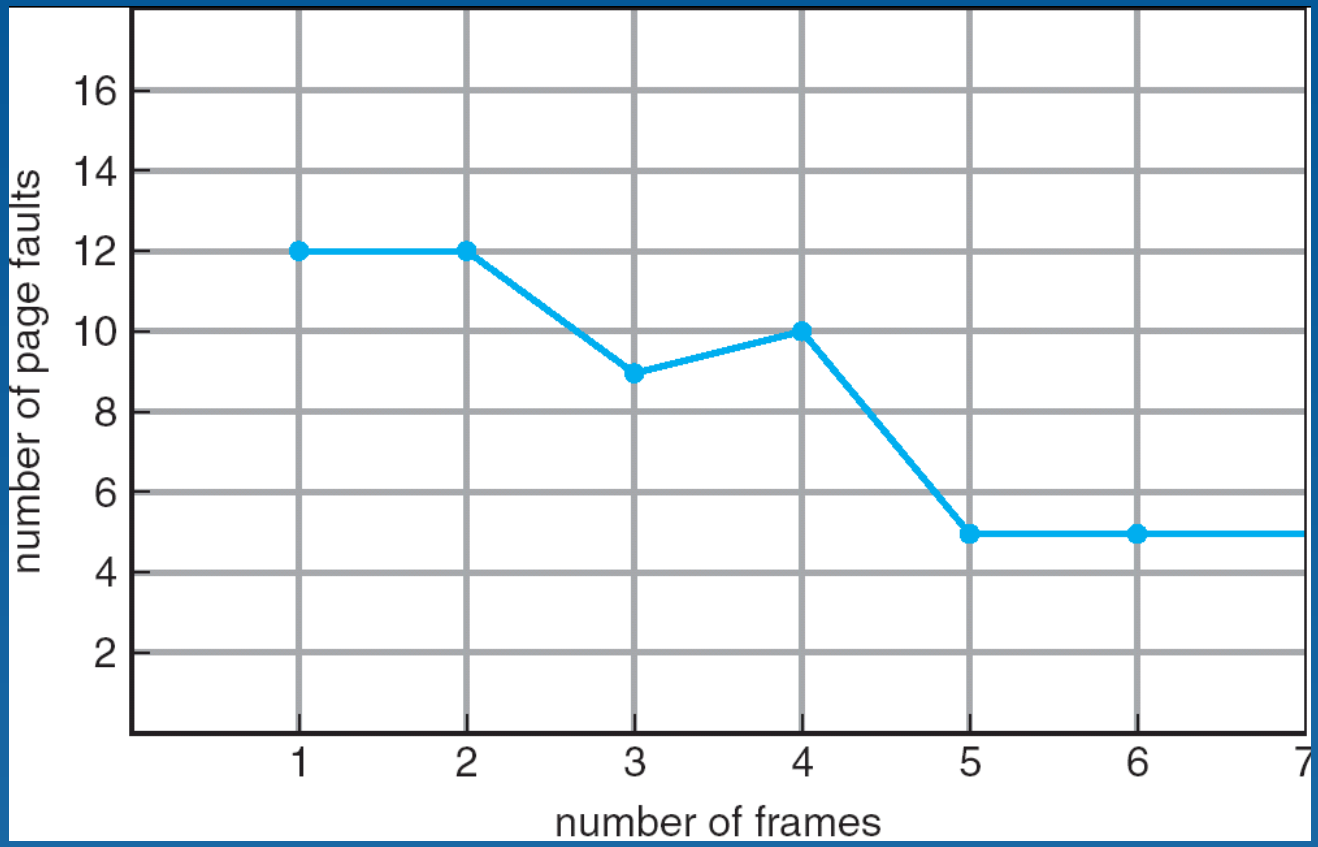


缺页 10 次

- Belady's Anomaly

物理页帧越多 ⇒ 缺页次数反而增加

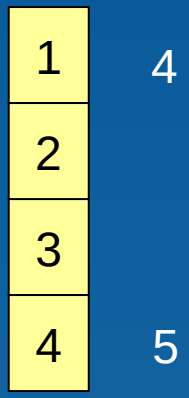
FIFO 算法存在 Belady's Anomaly



Optimal 算法

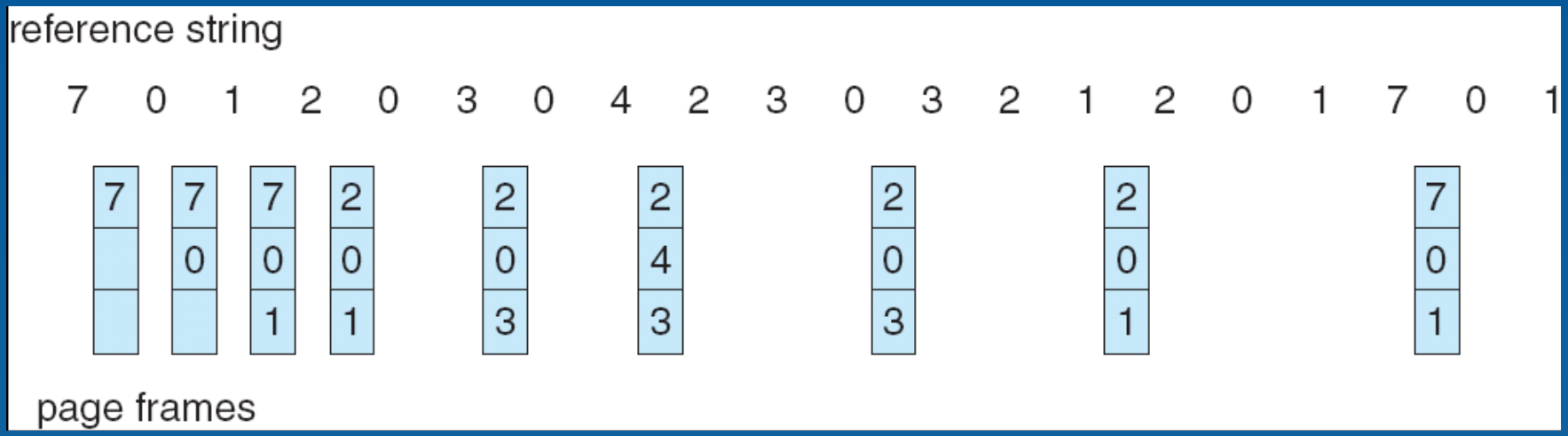
- 置换掉最长时间不被引用的页面

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



缺页 6 次

Optimal 算法



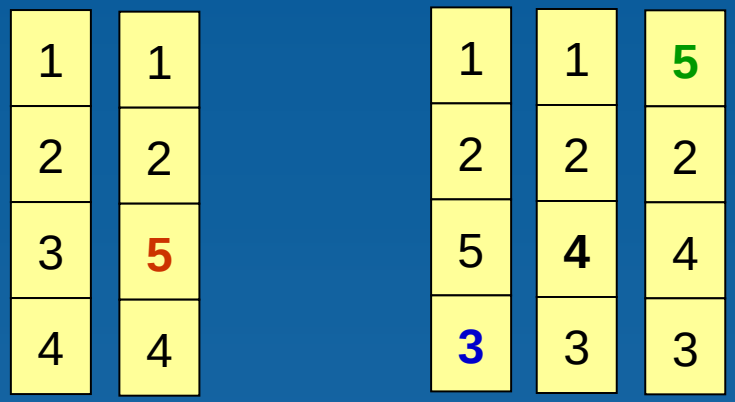
Optimal 算法

- 怎么知道“最长时间不被引用”？
- 现实不可行
- 但是，此算法可用来对比其它算法

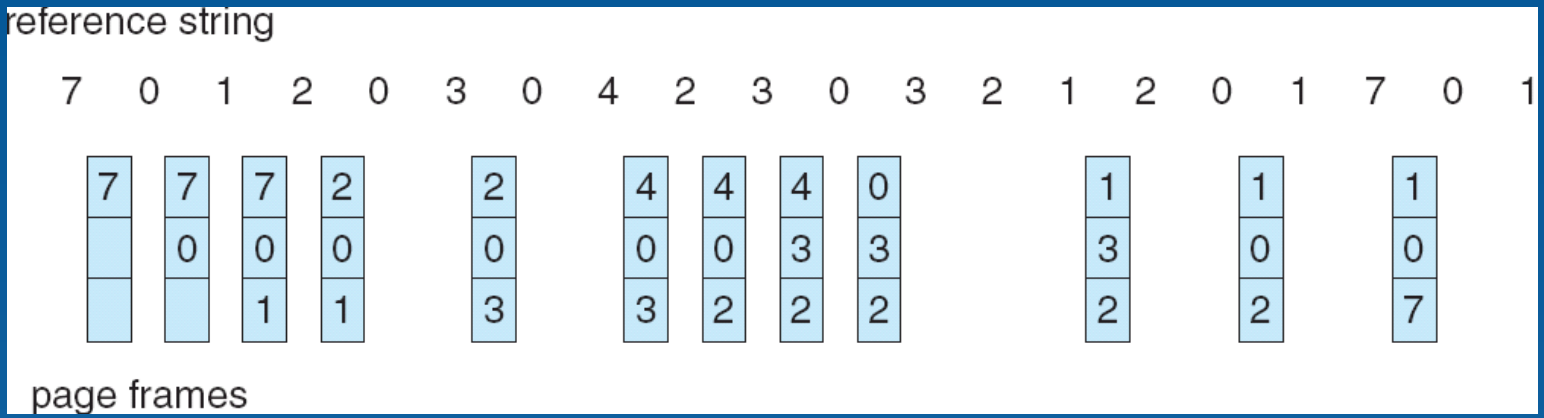
Least Recently Used (LRU) 算法

- 引用串

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



最近最少使用 (LRU) 算法



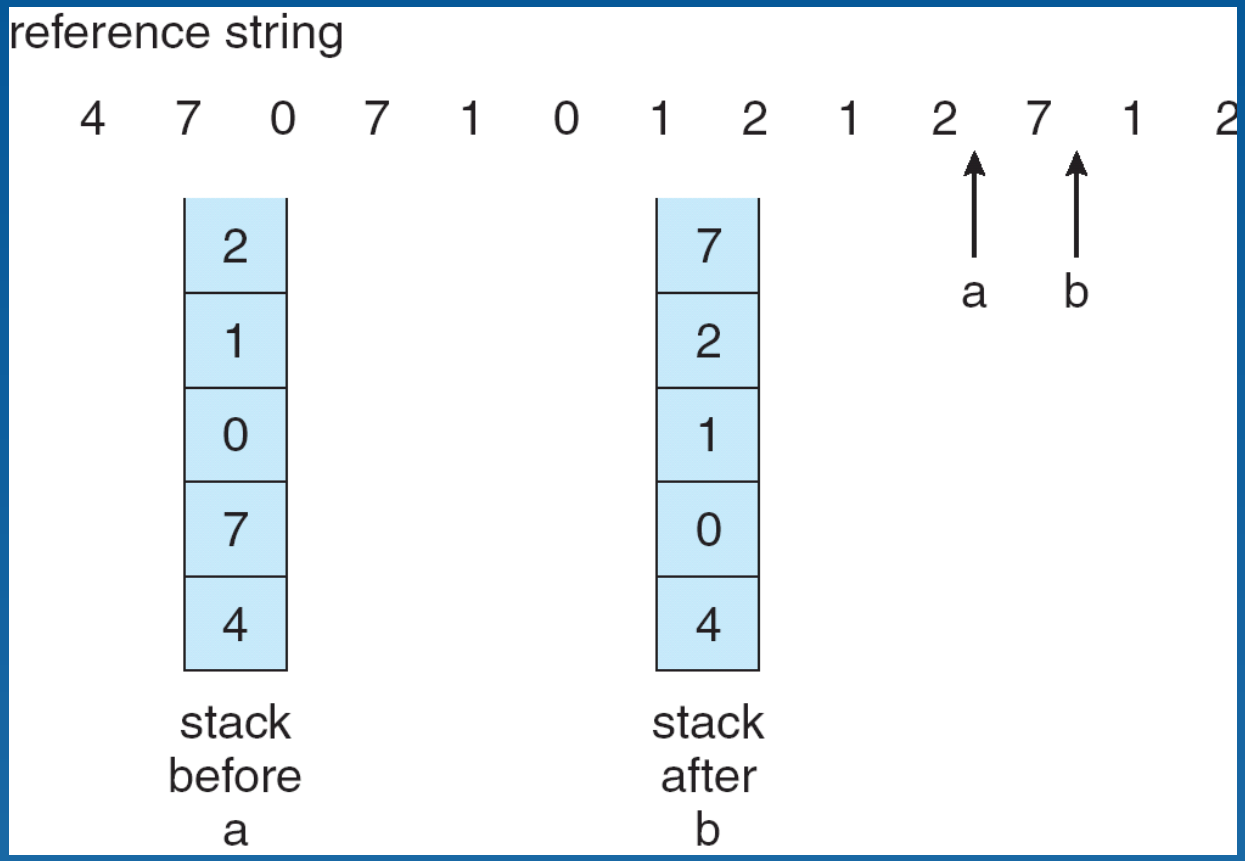
LRU算法，如何获知“多长时间没引用”？

- 利用计数器实现
  - 每个页表项带一计数器
  - 每次引用页面时，页表项的计数器更新为当时的时钟值
  - 调用置换算法时，选取计数器最“早”的页面，换出

LRU 算法，如何获知“多长时间没引用”？

- 利用堆栈实现 – 设计双向链表维护一个堆栈
  - 引用某页面时
    - 将该页面移动至栈顶
  - 页面置换时，不需要搜索、选取页面

LRU 算法，如何获知“多长时间没引用”？



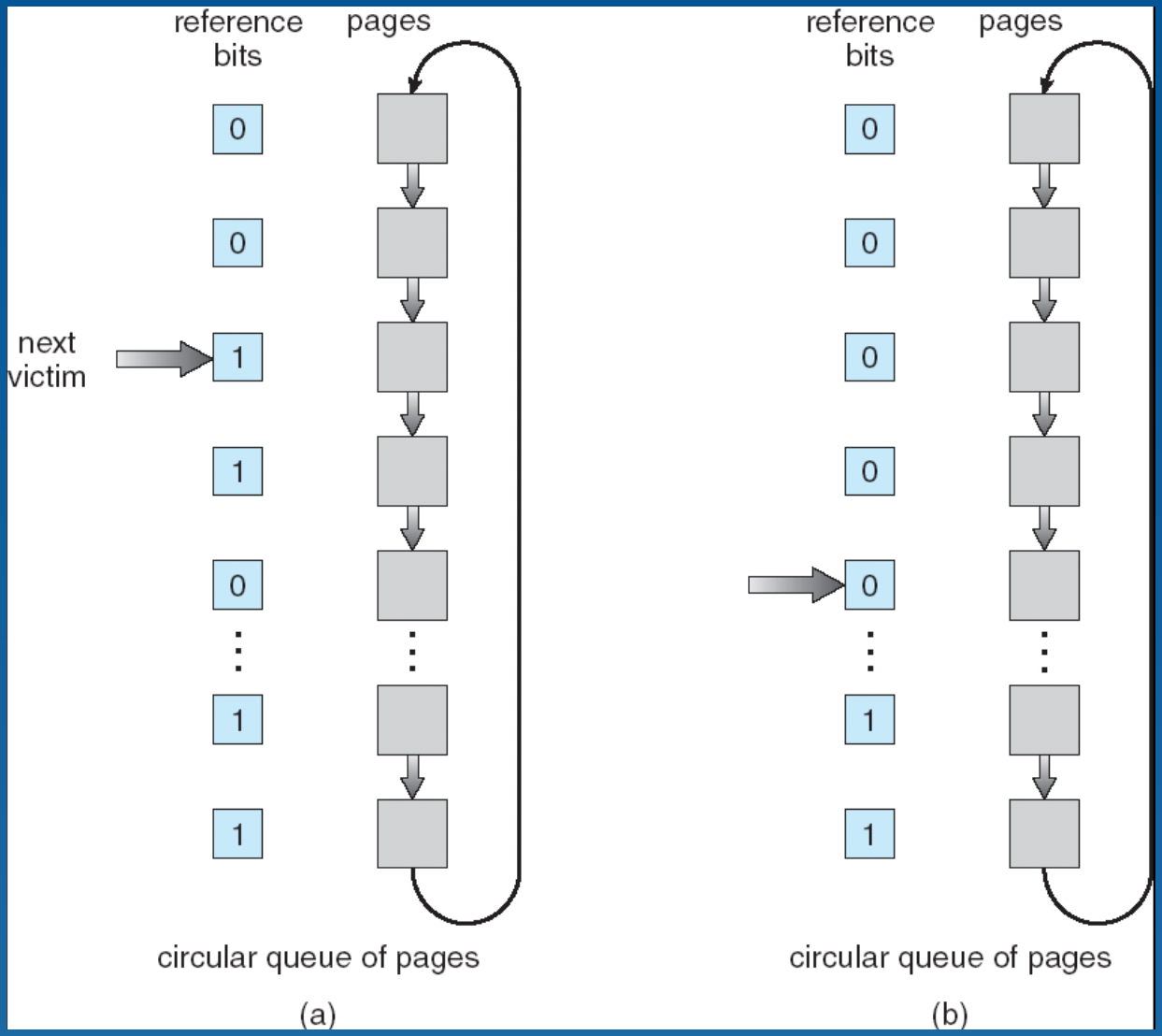
近似 LRU 算法

- 引用位 (Reference bit)
  - 每个页表项设计 1 位，引用位
  - 初始时 = 0
  - 页面被访问时，引用位被置 1
  - 需要置换页面时，总是选取引用位为 0 的页面（如果存在这样的页面）
  - 近似算法不规定所有引用位为 0 的页面的顺序

近似 LRU 算法

- Second chance 算法，也称 clock 算法
  - 也需要引用位
  - 需要置换页面时，（顺时针顺序）考察下一页面。如果引用位是 0 ，则选中
  - 如果其引用位 = 1
    - 引用位被置 0
    - 留在内存，这次不选它
    - 考察下一页面（按照顺时针顺序），重复上述程序判断引用位

clock 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)

计数 (Counting) 算法

- 每个页面附着一个计数器。计数器记录对该页面的引用次数
- **LFU 算法：** 计数值最小的页面被选中、换出
- **MFU 算法：** 计数值最大的页面被选中。基于这样的假设：计数值最小的页面可能刚刚装入内存，因此还来不及引用。


End