第9章 虚拟内存

内容

- 1、背景
- 2、按需调页
- 3、页面置换
- 4、页框分配
- 5、颠簸
- 6、系统内存分配
- 7、其它考虑

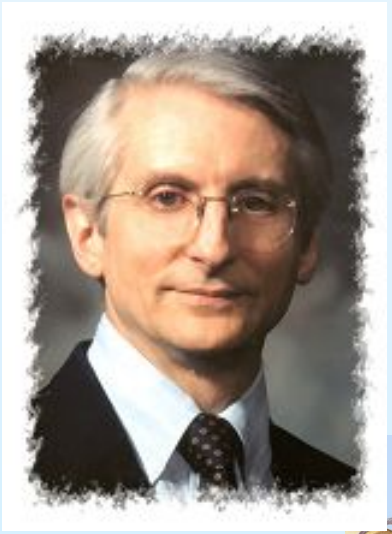
1、背景

背景

- 代码必须装入内存才能执行，但是并不是所有代码必须全部装入内存
  - 错误代码
  - 不常用的函数
  - 大的数据结构
- 局部性原理：一个程序只要部分装入内存就可以运行
  - 整个程序不是同一时间都要运行
- 程序部分装入技术优点
  - 进程大小不再受到物理内存大小限制，用户可以在一个虚拟的地址空间编程，简化了编程工作量
  - 每个进程需要的内存更小，因此更多进程可以并发运行，提供了CPU的利用率
  - I/O更少（载入的内容更少），用户程序运行更快

局部性原理

- 1968年，Denning指出：程序在执行时将呈现出**局部性规律**，即在一较短的时间内，程序的执行仅局限于某个部分；相应地，它所访问的存储空间也局限于某个区域
  - 程序执行时，除了少部分的转移和过程调用外，在大多数情况下仍然是顺序执行的
  - 过程调用将会使程序的执行轨迹由一部分区域转至另一部分区域，过程调用的深度一般小于5。程序将会在一段时间内都局限在这些过程的范围内运行
  - 程序中存在许多循环结构，多次执行
  - 对数据结构的处理局限于很小的范围



虚拟内存

- 虚拟存储技术：当进程运行时，先将其一部分装入内存，另一部分暂留在磁盘，当要执行的指令或访问的数据不在内存时，由操作系统自动完成将它们从磁盘调入内存执行。

- 虚拟地址空间：分配给进程的虚拟内存
- 虚拟地址：在虚拟内存中指令或数据的位置
- 虚拟内存：把内存和磁盘有机结合起来使用，得到一个容量很大的“内存”，即虚存
- 虚存是对内存的抽象，构建在存储体系之上，由操作系统来协调各存储器的使用

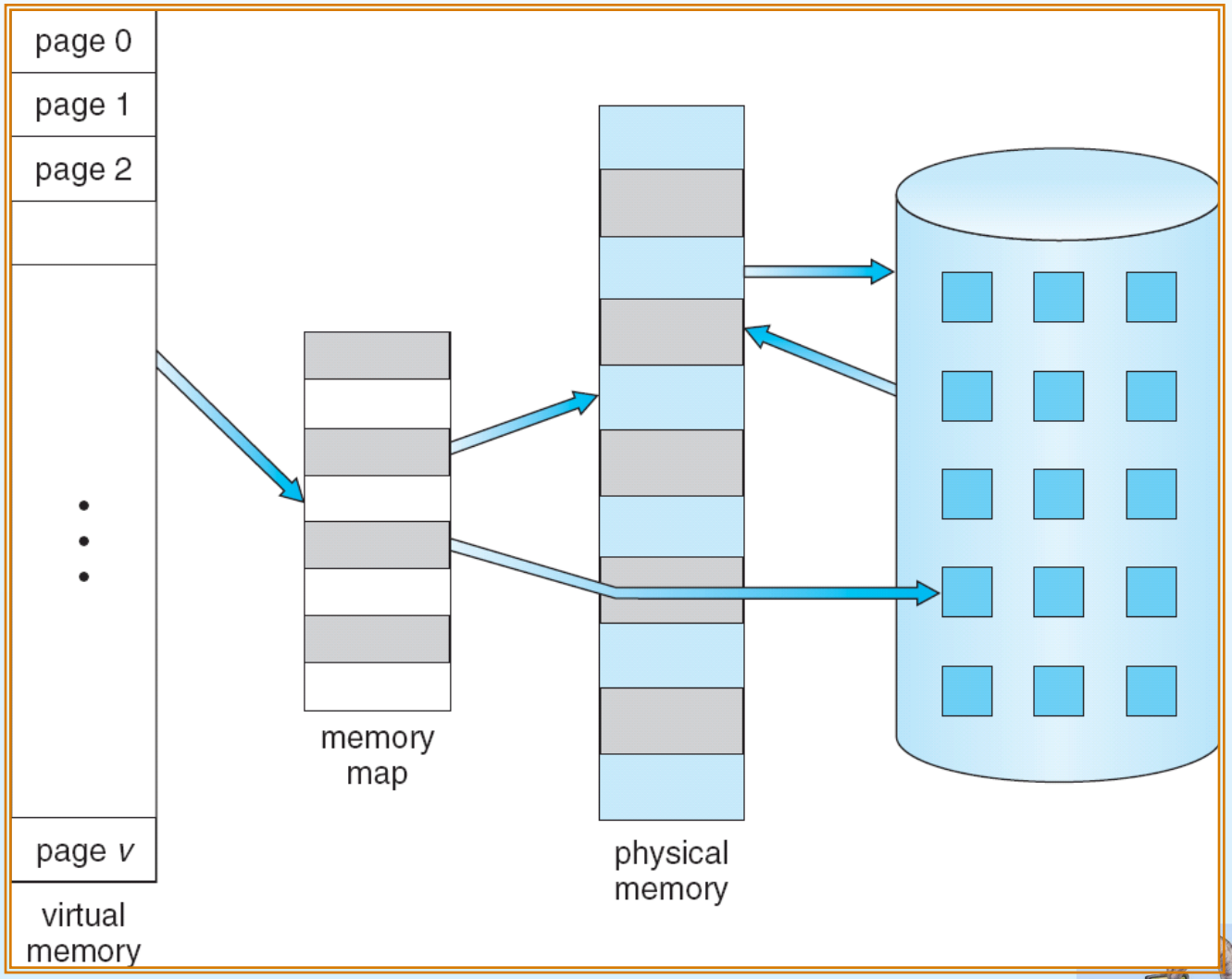
虚拟内存

- 虚拟内存—区分开物理内存和用户逻辑内存
  - 只有部分运行的程序需要在内存中
  - 逻辑地址空间能够比物理地址空间大
  - 必须允许页面能够被换入和换出
  - 允许更有效的进程创建

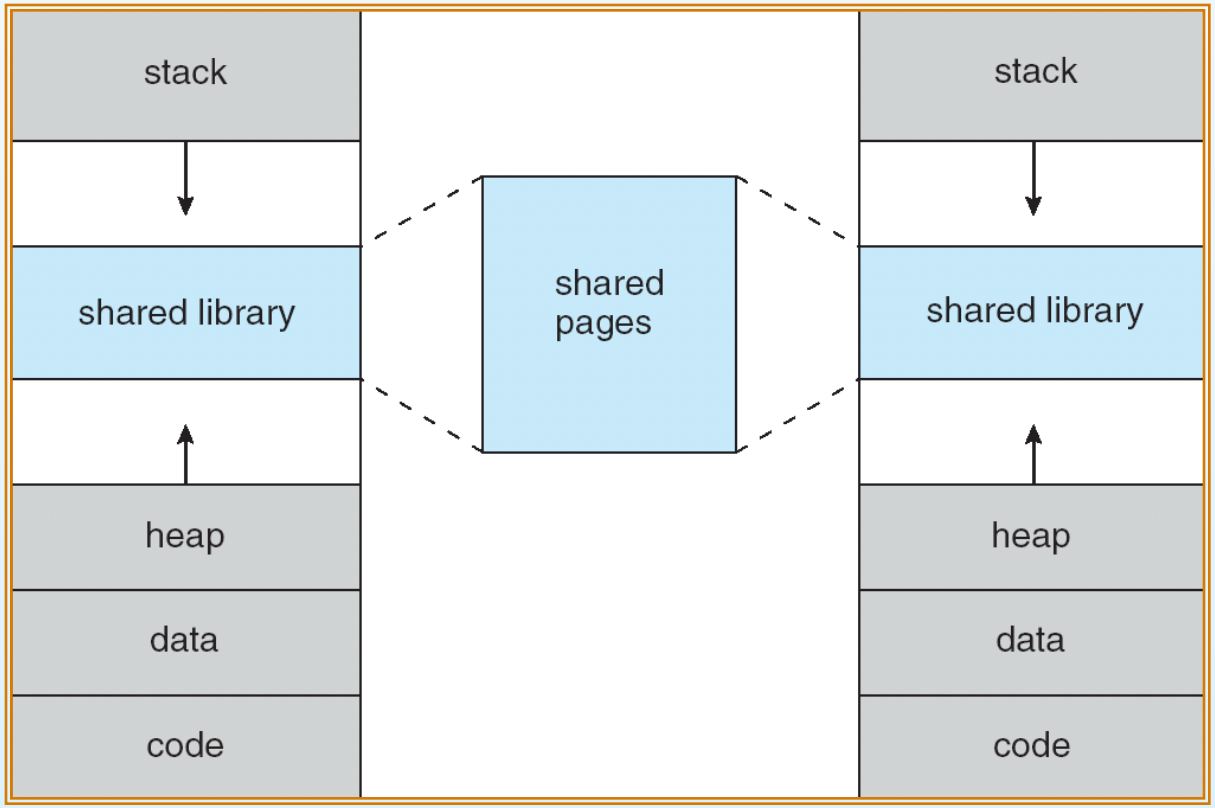
虚拟内存大于物理内存

虚拟存储器的大小由2个因素决定

1、操作系统字长
2、内存外存容量



使用虚拟内存的共享库



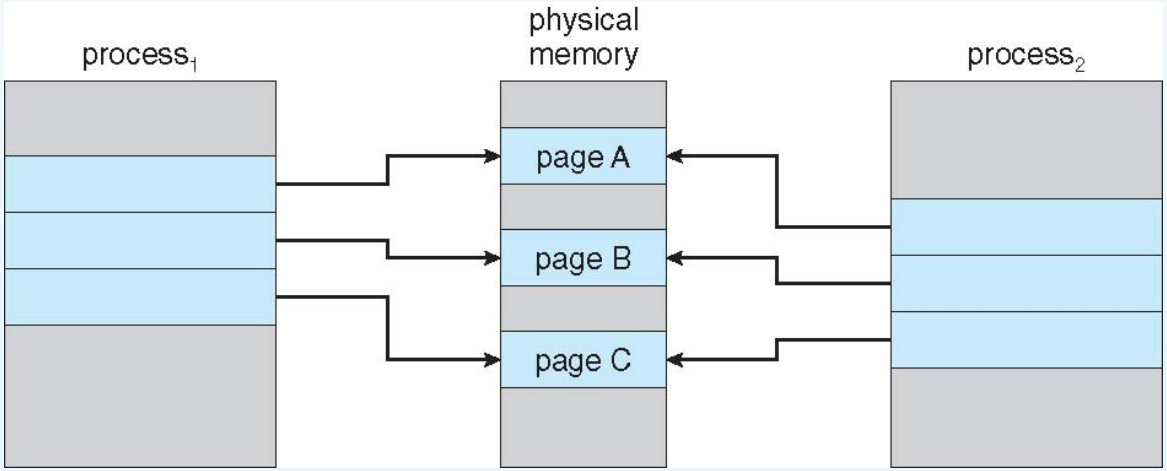
- 通过将共享对象映射到虚拟地址空间，系统库可用被多个进程共享
- 虚拟内存允许进程共享内存
- 虚拟内存可允许在创建进程期间共享页，从而加快进程创建

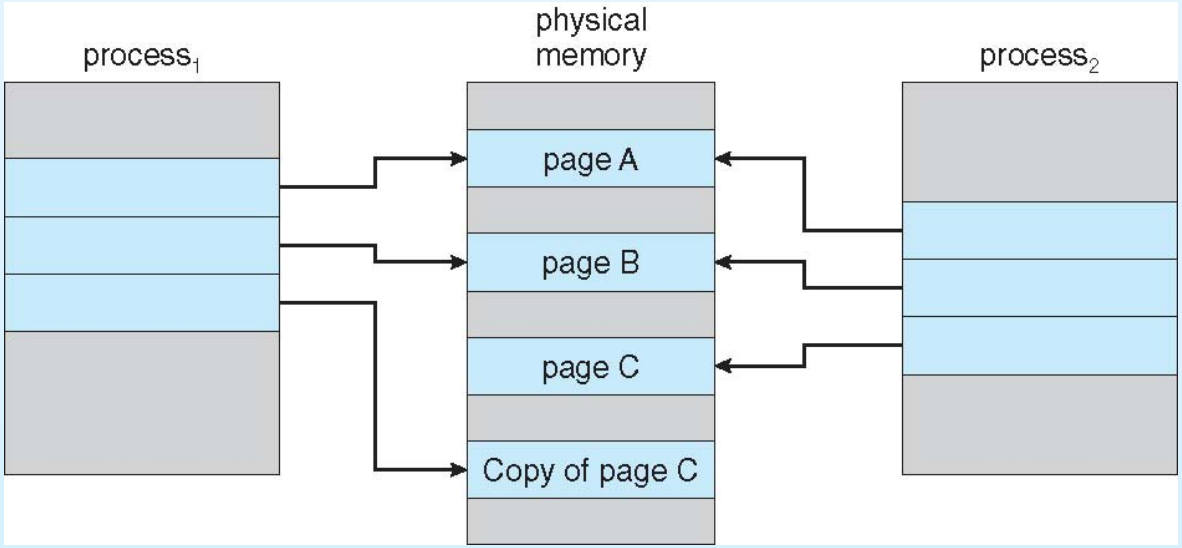
写时复制（Copy-on-Write）

- 写时复制允许父进程和子进程在初始化时共享页面
  - 如果其中一个进程修改了一个共享页面，会产生副本
  - 更加高效
  - 应用在Windows XP，Linux等系统

- `vfork：fork()`变形，不使用写时复制

写时复制例子


process1
physical memory
process2
page A
page B
page C


process1
physical memory
process2
page A
page B
page C
Copy of page C

虚拟内存的实现

- 虚拟内存能够通过以下手段来执行实现:
  - 虚拟页式（虚拟存储技术+页式存储管理）
  - 虚拟段式（虚拟存储技术+段式存储管理）

- 虚拟页式有两种方式
  - 请求分页（ Demand paging ）
  - 预调页（Prepaging）

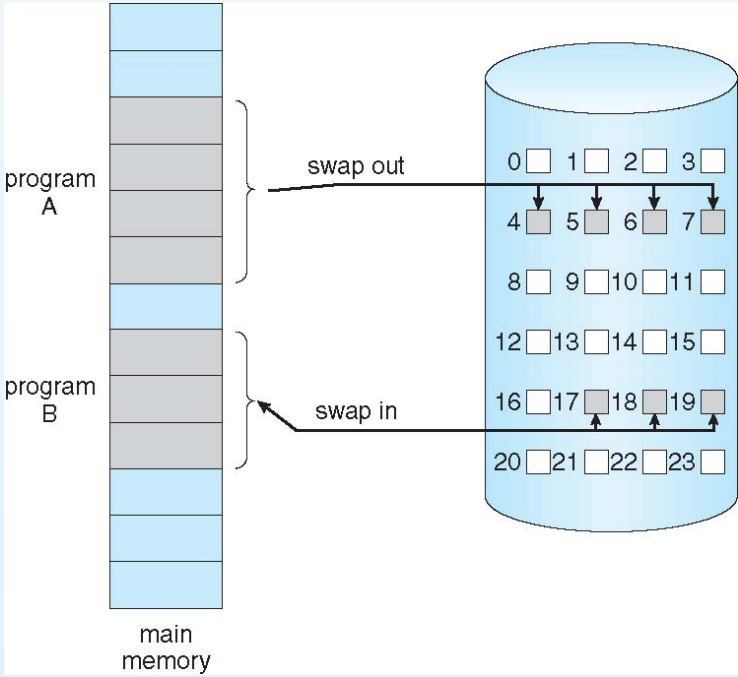
2、请求分页

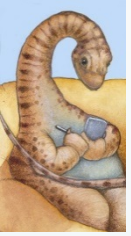
虚拟页式存储管理

- 基本思想
  - 进程开始运行之前，不是装入全部页面，而是装入一个或零个页面
  - 运行之后，根据进程运行需要，动态装入其他页面
  - 当内存空间已满，而又需要装入新的页面时，则根据某种算法置换内存中的某个页面，以便装入新的页面

请求分页（按需调页）

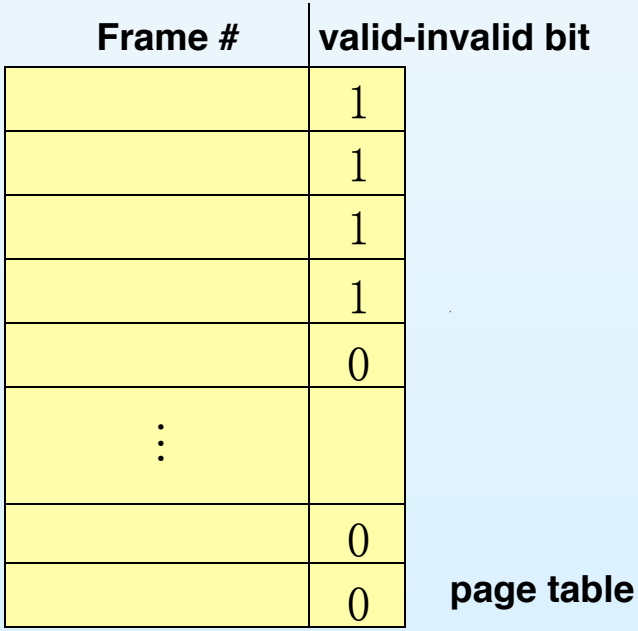
- 只有在一个页需要的时候才把它换入内存
  - 需要很少的I/O
  - 需要很少的内存
  - 快速响应
  - 支持多用户
- 类似交换技术，粒度不同
  - 交换程序（swapper）对整个进程进行操作
  - 调页程序（pager）只是对进程的单个页进行操作
- 需要页⇒ 查阅此页
  - 无效的访问 ⇒ 中止
  - 不在内存 ⇒ 换入内存
- 懒惰交换
  - 只有在需要页时，才将它调入内存





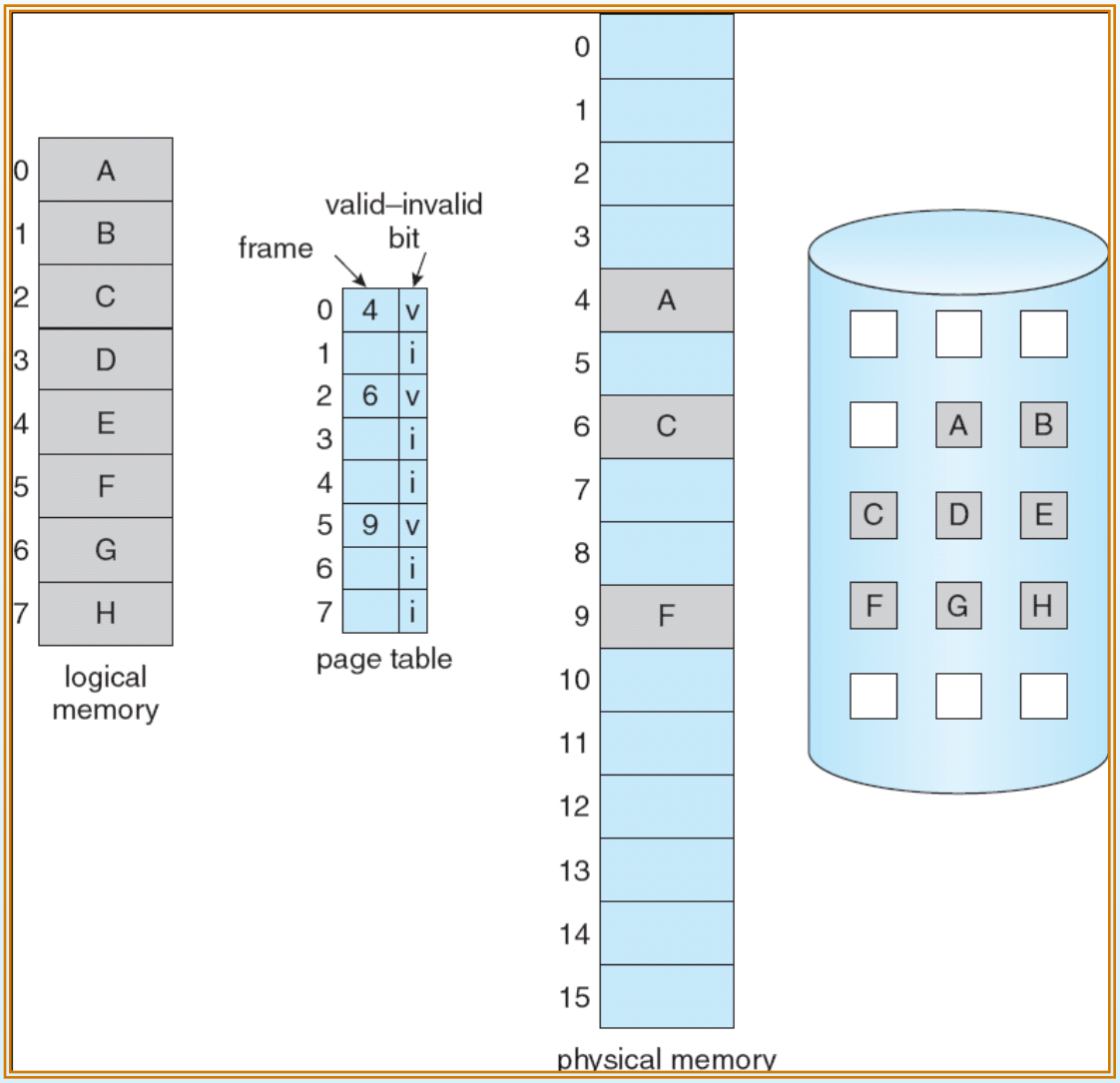
有效-无效位(Valid-Invalid)

- 在每一个页表的表项有一个有效- 无效位相关联，1表示在内存，0表示不内存
- 在所有的表项，这个位被初始化为0
- 一个页表映象的例子



- 在地址转换中，如果页表表项位的值是0 ⇒缺页中断（ page fault ）

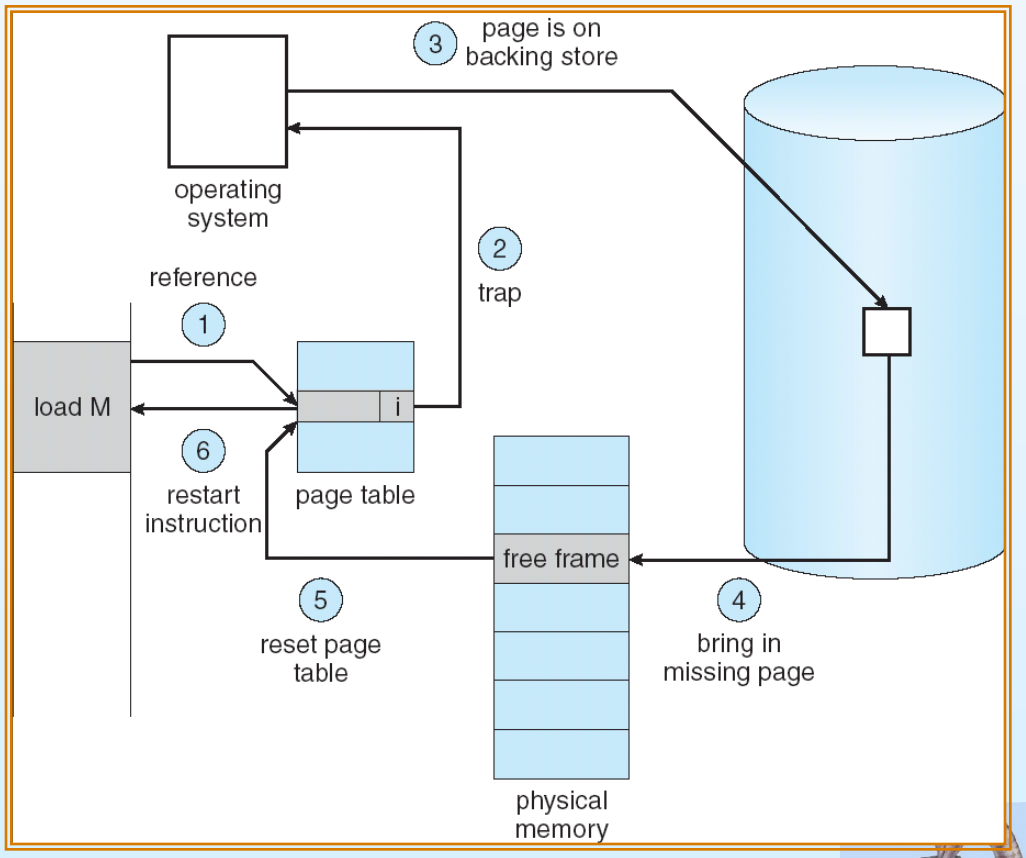
有页不在内存的页表


0
1
2
3
4
5
6
7
A
B
C
D
E
F
G
H
logical memory
valid–invalid bit
frame
0 4 v
1 i
2 6 v
3 i
4 i
5 9 v
6 i
7 i
page table
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
A
C
F
physical memory
A
B
C
D
E
F
G
H

缺页中断（页错误）

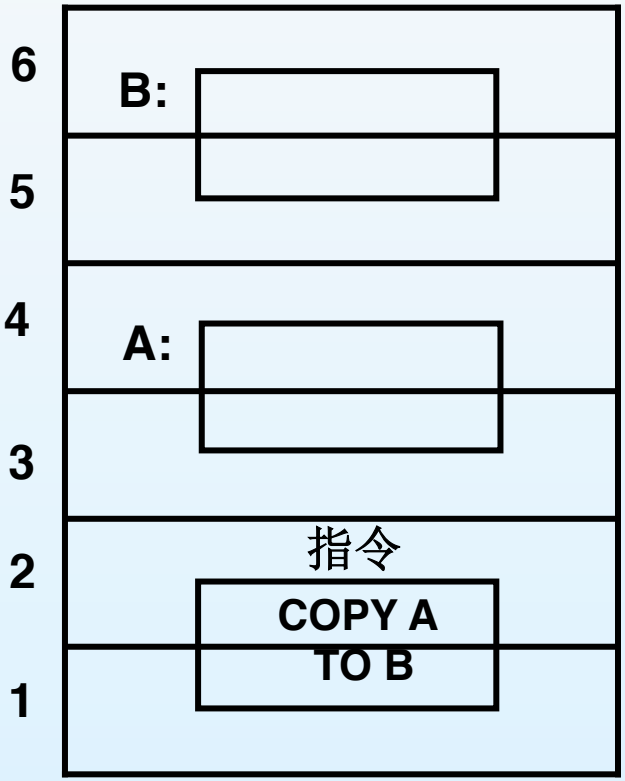
- 如果对一个页的访问，首次访问该页需要陷入OS ⇒ 缺页中断

1. 访问指令或数据
   - 发现有效无效位为0
2. 查看另一个表来决定
   - 无效引用 ⇒ 终止
   - 仅仅不在内存
3. 找到页在后备存储上的位置
4. 得到空闲帧，把页换入帧
5. 重新设置页表，把有效位设为v
6. 重启指令： 最近未使用



请求分页

- 极端情况：进程执行第一行代码时，内存内没有任何代码和数据
  - 进程创建时，没有为进程分配内存，仅建立PCB
  - 导致缺页中断
  - 纯请求分页
- 一条指令可能导致多次缺页（涉及多个页面）
  - 幸运的是，程序具有局部性（locality of reference）
- 请求分页需要硬件支持
  - 带有效无效位的页表
  - 交换空间
  - 指令重启



请求分页的性能

- 缺页率：$0 \le p \le 1.0$
  - 如果 $p = 0$，没有缺页
  - 如果 $p = 1$， 每次访问都缺页
- 有效存取时间（ EAT ）

  $$EAT = (1 - p) \times \text{内存访问时间} + p \times \text{页错误时间}$$
- 页错误时间（包含多项处理的时间，主要有三项）
  - 处理缺页中断时间
  - 读入页时间
  - 重启进程开销
  - [页交换出去时间] （不是每次都需要）

一个请求分页的例子

- 存取内存的时间= 200 nanoseconds (ns)
- 平均缺页处理时间 = 8 milliseconds (ms)
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800

- 如果每1,000次访问中有一个页错误，那么

  EAT = 8.2 us

这是导致计算机速度放慢40倍的影响因子！

请求分页性能优化

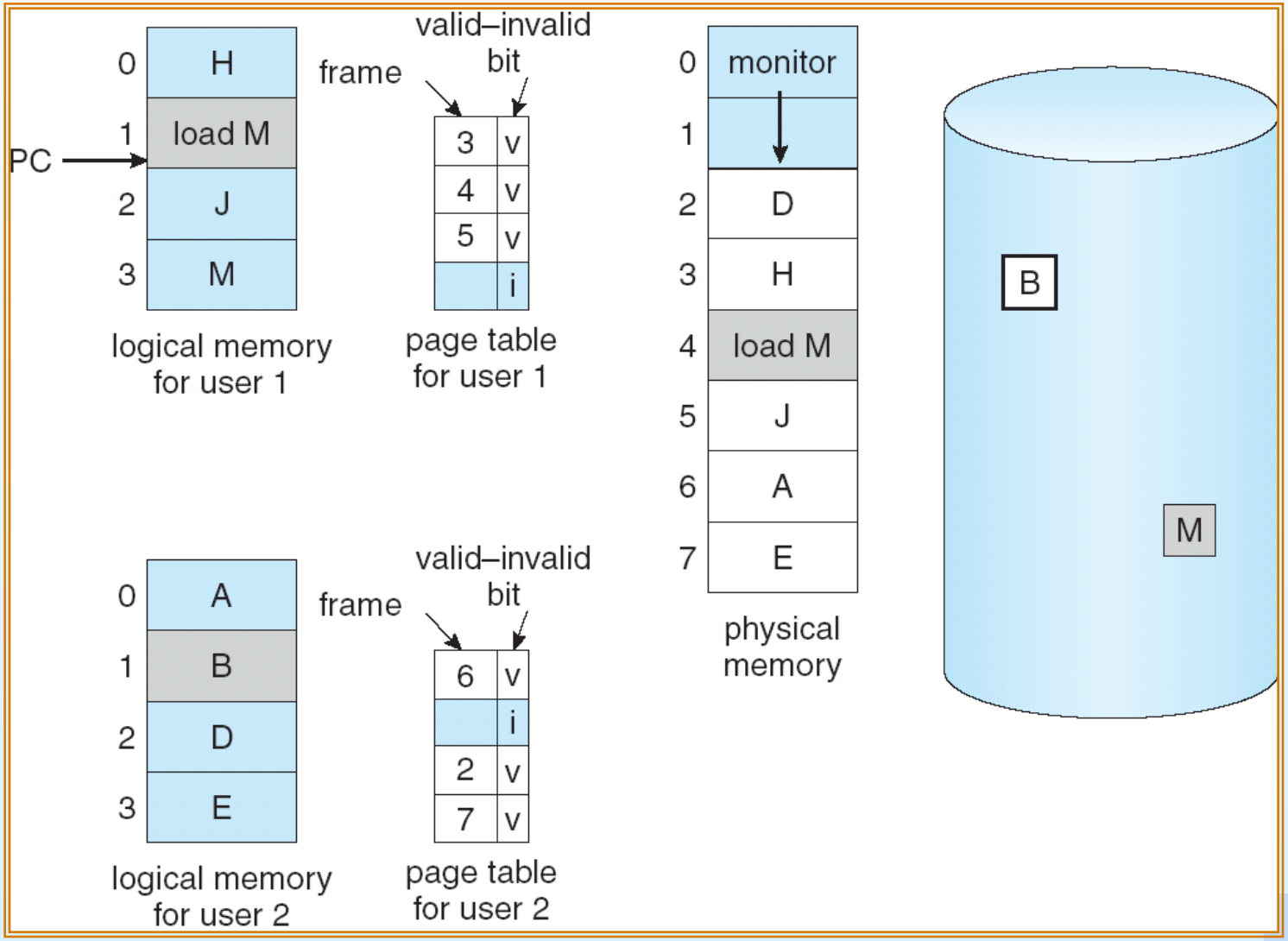
- 页面转换时采用交换空间，而不是文件系统
  - 交换区的块大，比文件系统服务快速
- 在进程装载时，把整个进程拷贝到交换区
  - 基于交换区调页
  - 早期的 BSD Unix
- 利用文件系统进行交换
  - Solaris和当前的BSD Unix
  - 部分内容仍旧需要交换区（堆栈等）

3、页面置换

如果没有空闲页怎么办？

- 解决方法：
  - 终止进程
  - 交换进程
  - 页面置换（page replacement），又称页置换、页淘汰
- 页置换
  - 找到内存中并没有使用的一些页，换出
  - 算法
  - 性能 - 找出一个导致最小缺页数的算法
  - 同一个页可能会被装入内存多次
- **页面置换**
  - 通过修改缺页服务例程，来包含页面置换，防止分配过多
  - 修改（脏）位*modify* (*dirty*) *bit*来防止页面转移过多—只有被修改的页面才写入磁盘
  - 页置换完善了逻辑内存和物理内存的划分—在一个较小的物理内存基础之上可以提供一个大的虚拟内存

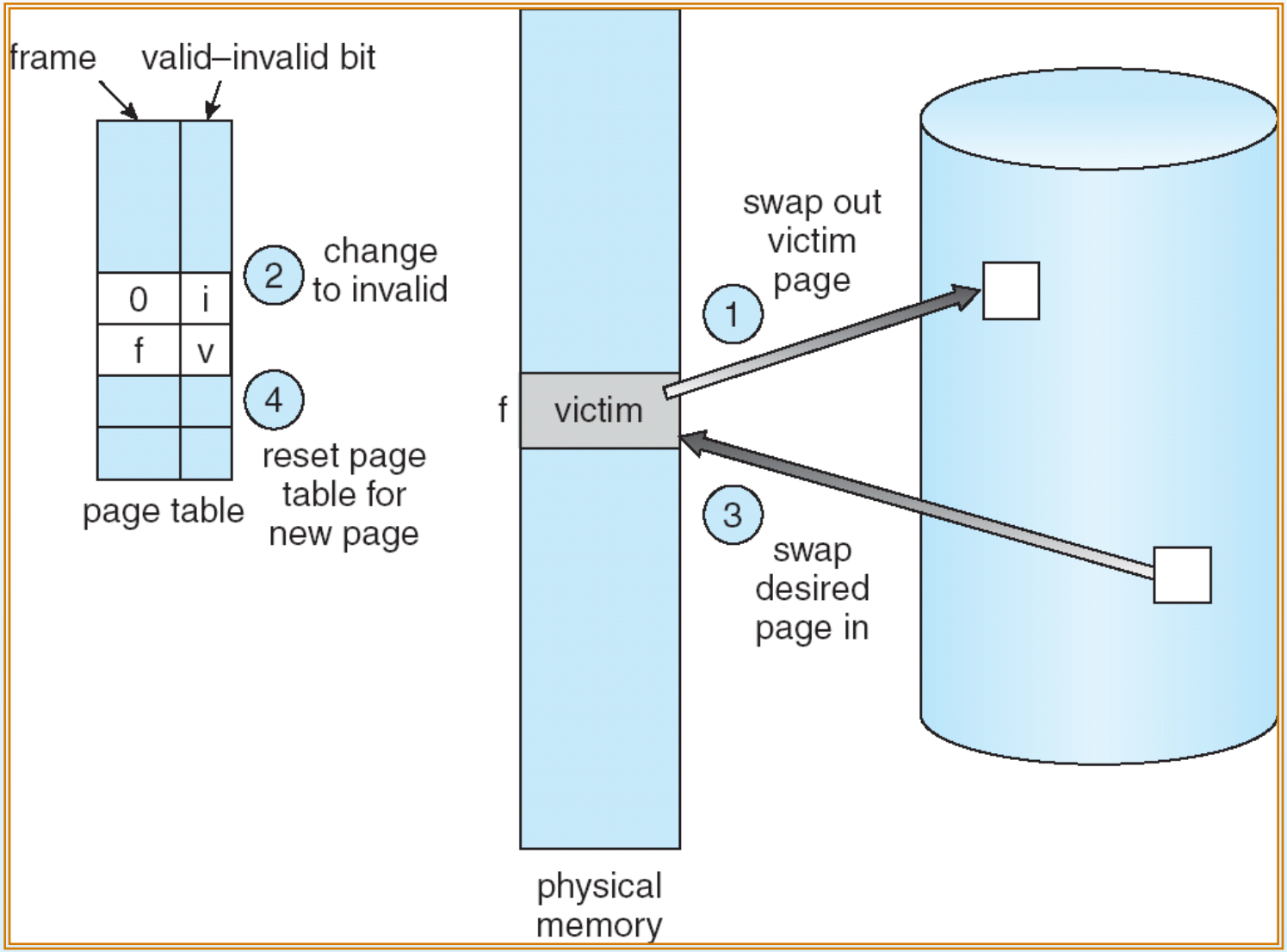
需要页置换的情况


0
1
2
3
H
load M
J
M
PC
logical memory
for user 1
frame
valid–invalid
bit
3 v
4 v
5 v
i
page table
for user 1
0
1
2
3
4
5
6
7
monitor
D
H
load M
J
A
E
physical
memory
B
M
0
1
2
3
A
B
D
E
logical memory
for user 2
frame
valid–invalid
bit
6 v
i
2 v
7 v
page table
for user 2

基本页置换方法

1. 查找所需页在磁盘上的位置
2. 查找一空闲页框
   - 如果有空闲页框，就使用它
   - 如果没有空闲页框，使用页置换算法选择一个“牺牲”页框（victim frame）
   - 将“牺牲”帧的内容写到磁盘上，更新页表和帧表
3. 将所需页读入（新）空闲页框，更新页表和帧表
4. 重启用户进程

页置换


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

页置换讨论

- 如果发生页置换，则缺页处理时间加倍
- 使用修改位（*modify bit*）或脏位（*dirty bit*）来防止页面转移过多—只有被修改的页面才写入磁盘
- 页置换完善了逻辑内存和物理内存的划分—在一个较小的物理内存基础之上可以提供一个大的虚拟内存

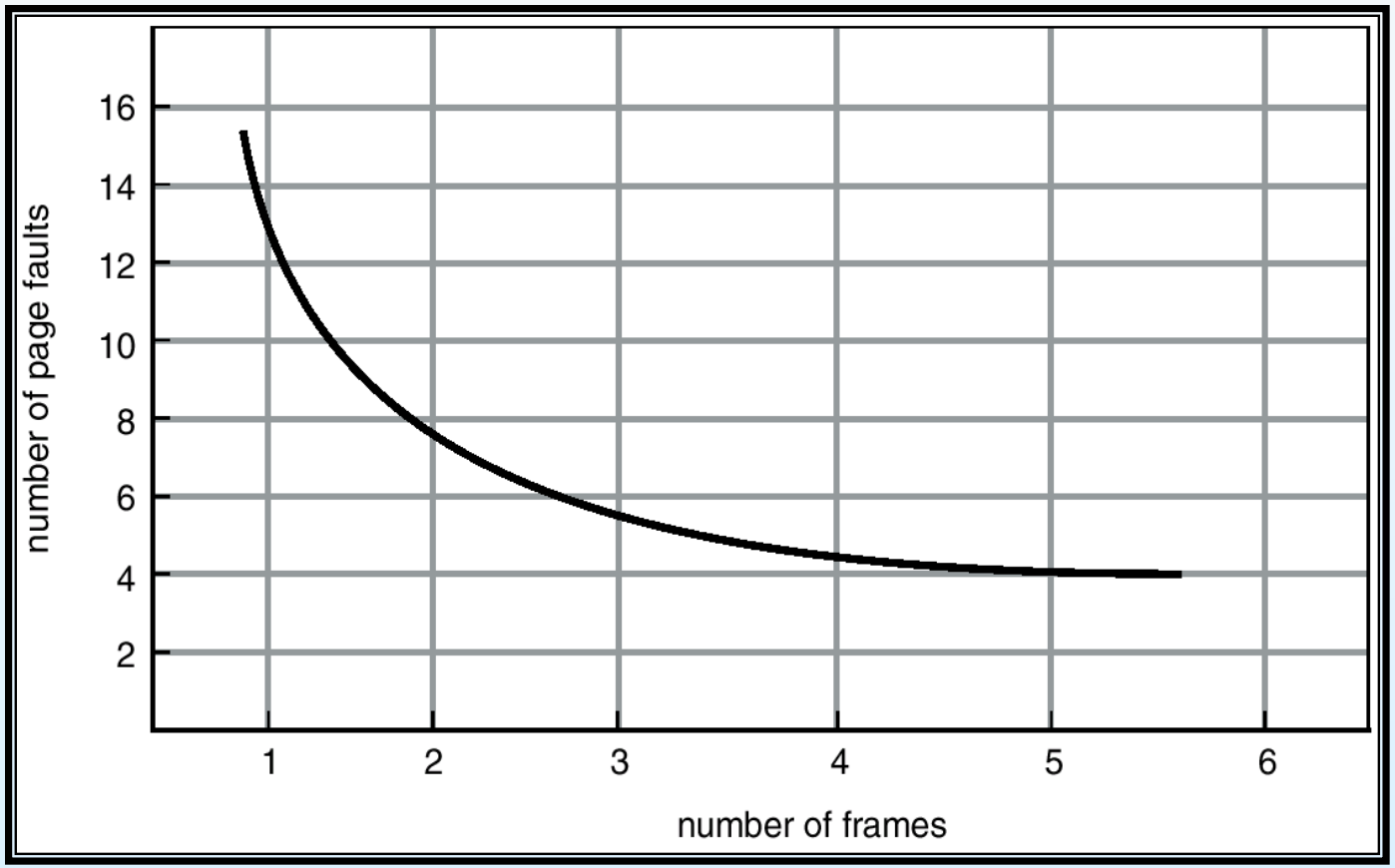
帧分配算法和页置换算法

- 为了实现请求调页，必须开发两个算法：
  - 如果在内存中有多个进程，那么帧分配算法决定为每个进程各分配多少帧
  - 当发生页置换时，页置换算法决定要置换的帧是哪一个

- 页面置换算法
  - **最小的缺页率**
  - 通过运行一个内存访问的特殊序列（访问序列），计算这个序列的缺页次数
  - 访问序列是

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

缺页与帧数量关系图


number of page faults
16
14
12
10
8
6
4
2
1
2
3
4
5
6
number of frames

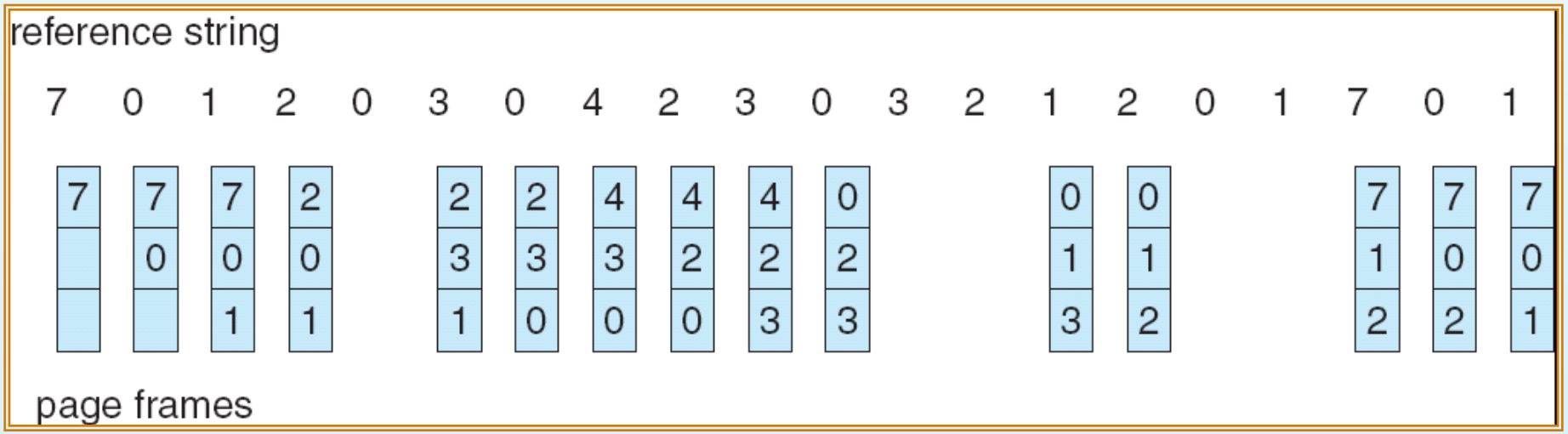
页面置换算法

- 最优置换置换算法（OPT）
- 先进先出置换算法（FIFO）
- 最近最少使用置换算法（LRU）
- 近似LRU算法
  - 二次机会法

- 要求：
  - 掌握设计思想、算法应用
  - 了解部分算法的实现

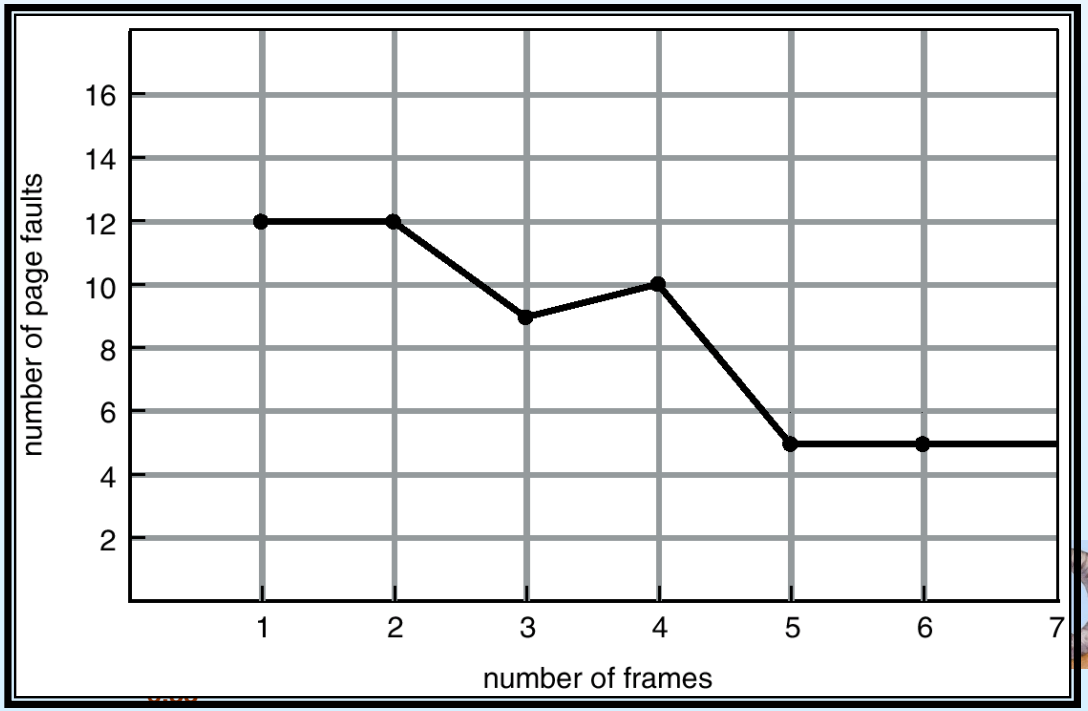
先进先出(FIFO)算法

- 置换在内存中驻留时间最长的页面
- 容易理解和实现、但性能不总是很好
- 实现：使用FIFO队列管理内存中的所有页



Belady异常

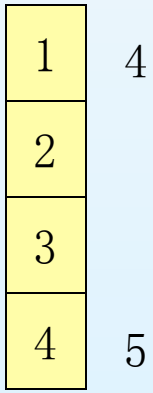
- 引用串 ：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3 个页框， 9次缺页
  - 4 个页框，10次缺页
- FIFO算法可能会产生Belady异常
  - 更多的页框 ⇒ 更多的缺页



最优置换算法

- 被置换的页是将来不再需要的或最远的将来不被使用的页
- 4 帧的例子

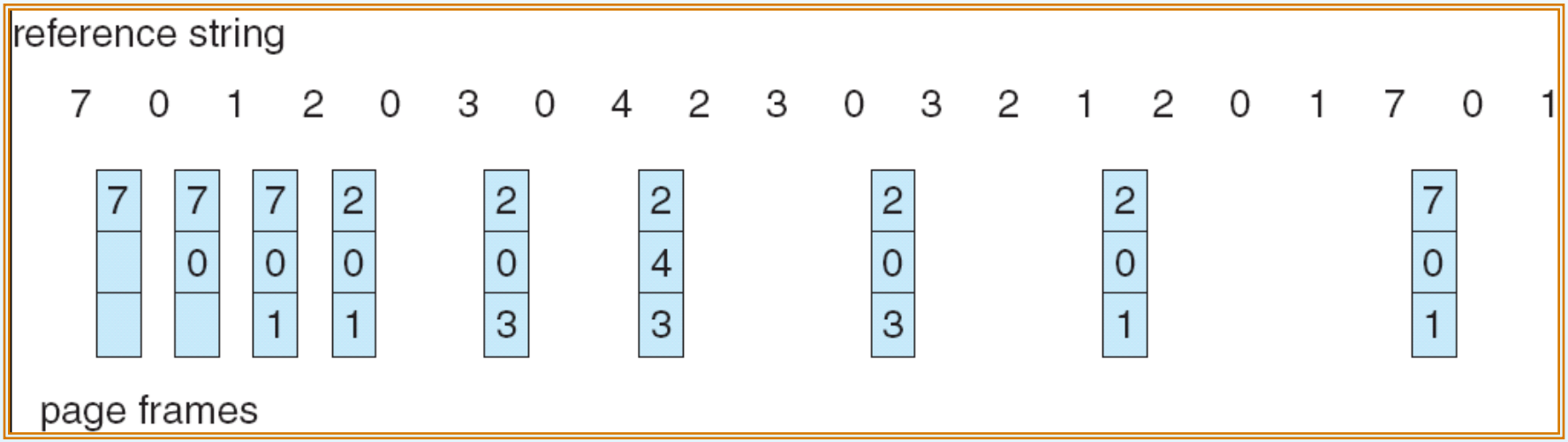
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

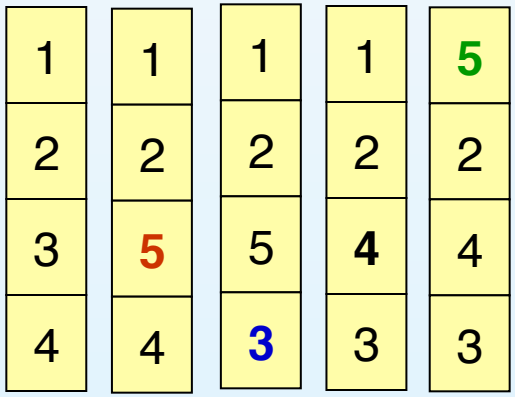
- 怎样知道的？
- 作用：作为一种标准衡量其他算法的性能

最优置换算法



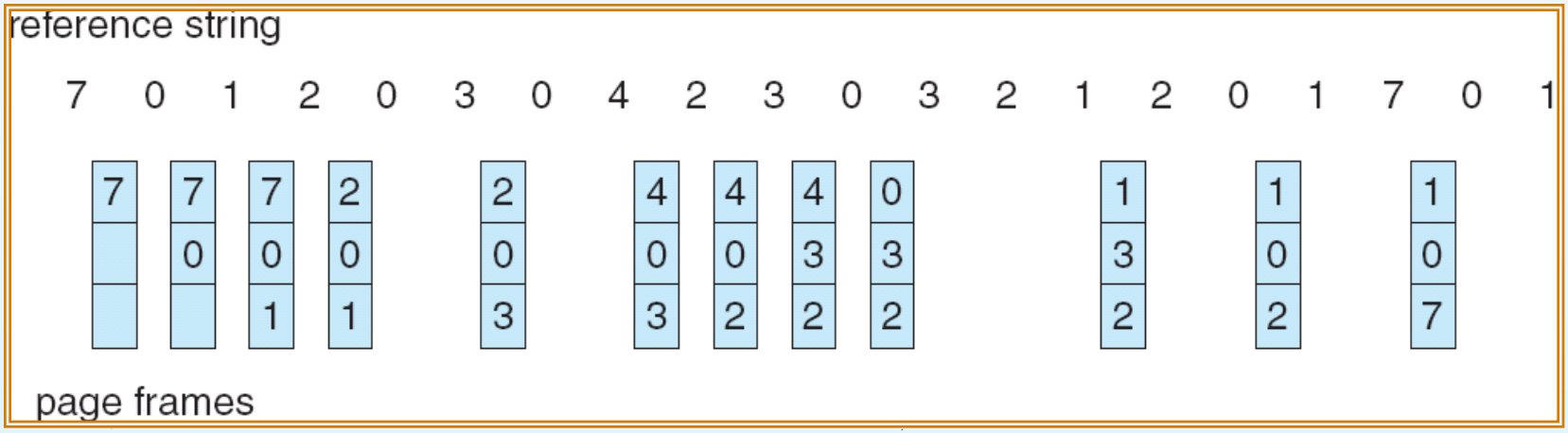
最近最少使用算法(LRU)

- 置换最长时间没有使用的页
- 性能接近OPT
- 引用串：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- 计数器的实现
  - 每一个页表项 有一个计数器（时间戳）或栈
  - 开销大，需要硬件支持

LRU 置換算法



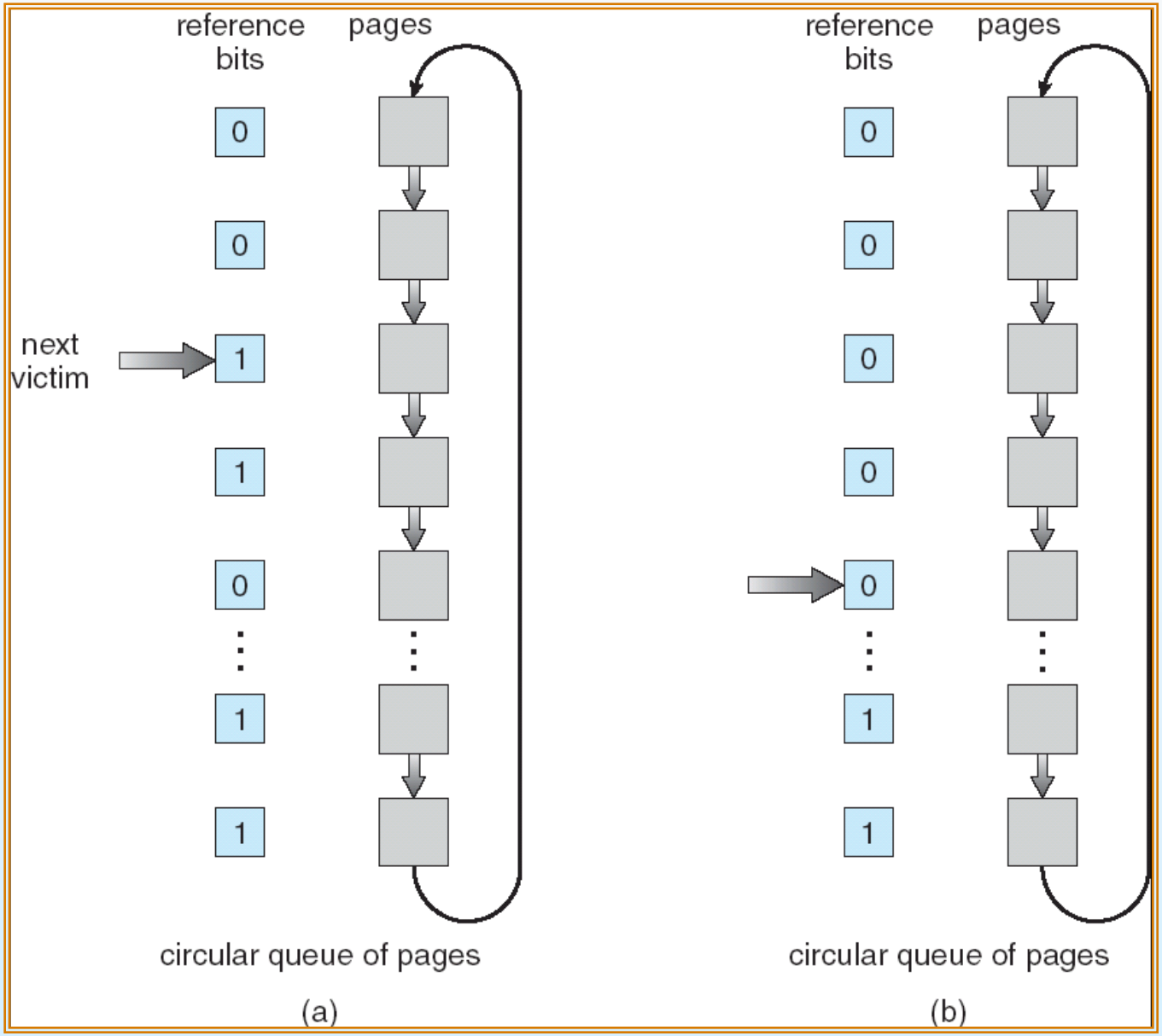
LRU近似算法

- LRU需要硬件支持
- LRU近似算法
- 引用位
  - 每个页都与一个位相关联r位，初始值位0
  - 当页访问时设位1
- 基于引用位的算法
  - 附加引用位算法
  - 二次机会算法
  - 增强型二次机会算法

LRU近似算法

- 附加引用位算法（LRU近似）
  - 为内存中的每个页设置一个8位字节
  - 在规定时间间隔内，把每个页的引用位转移到8位字节的高位，将其他位向右移一位，并舍弃最低位
  - 这8位移位寄存器包含最近8个时间周期内的页面使用情况
  - 最小值的页为最近最少使用页，可以被淘汰
- 二次机会算法（基本算法FIFO）
  - 需要引用位
  - 如果引用位为0，直接置换
  - 如果将要（以顺时针）交换的页访问位是1，则：
    - 把引用位设为0
    - 把页留在内存中
    - 以同样的规则，替换下一个页
  - 实现：时钟置换（顺时针方向，采用循环队列）

二次机会置换算法


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)