



L8. 虚拟内存



宋卓然

上海交通大学计算机系

songzhuoran@sjtu.edu.cn

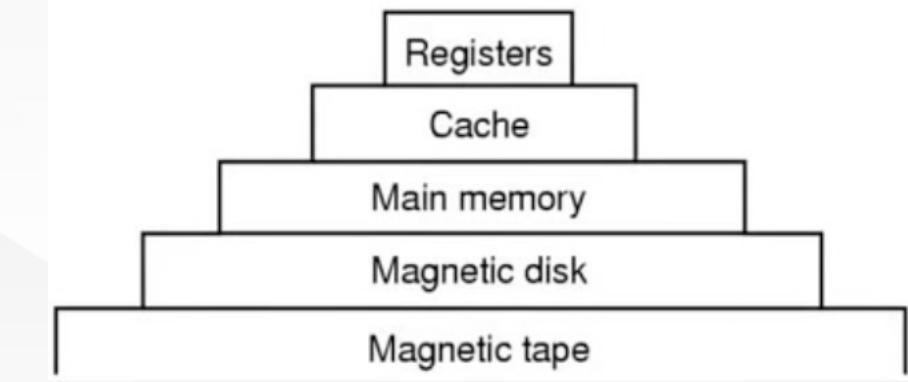
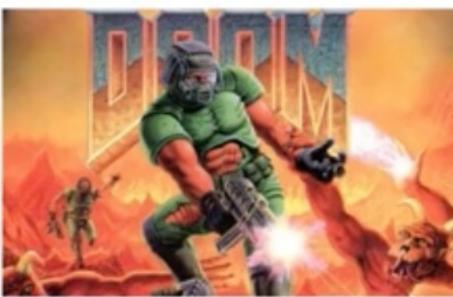


背景

- 程序很大，内存空间不够
- 金字塔型存储器层次结构
- 难以直接把程序放在硬盘上执行，速度太慢

电脑游戏

一代	二代	三代	四代	五代	六代	七代	八代
437K	883K	1.9M	6M	6.3M	59M	100M	138M



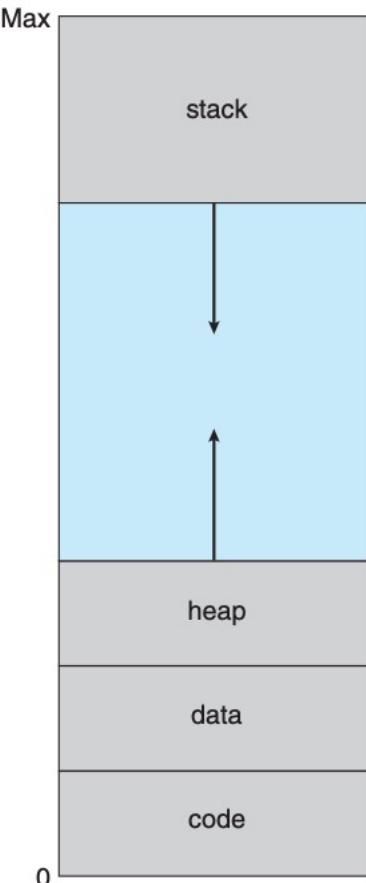
存储器层次结构





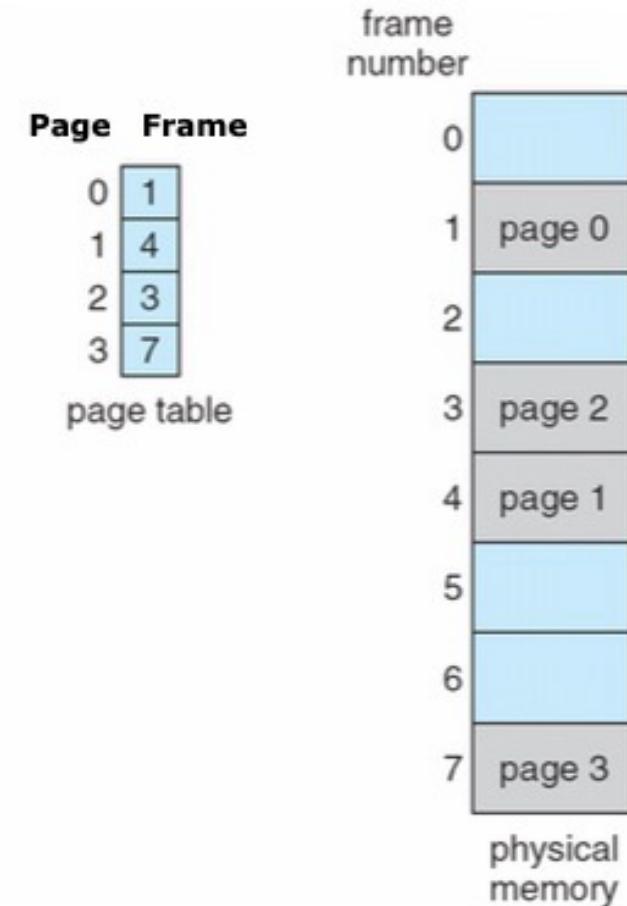
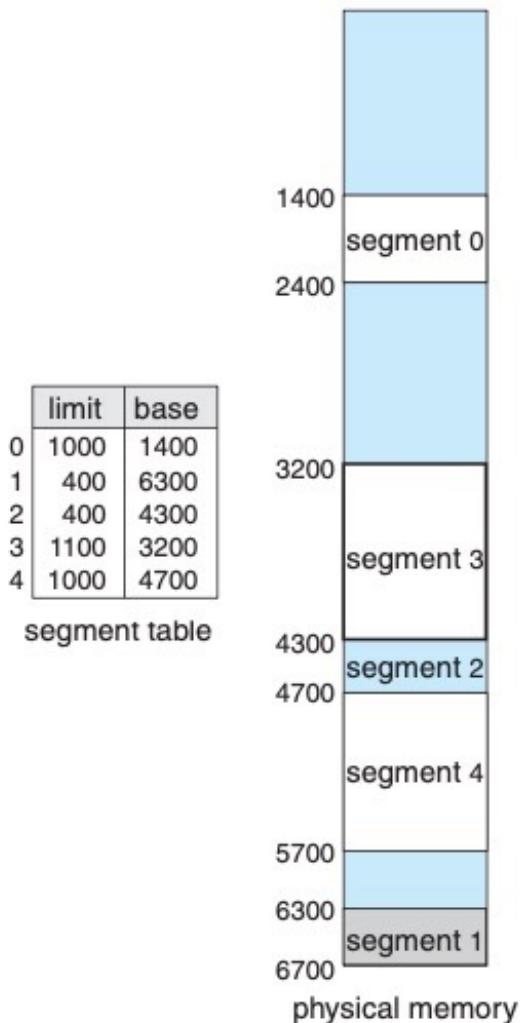
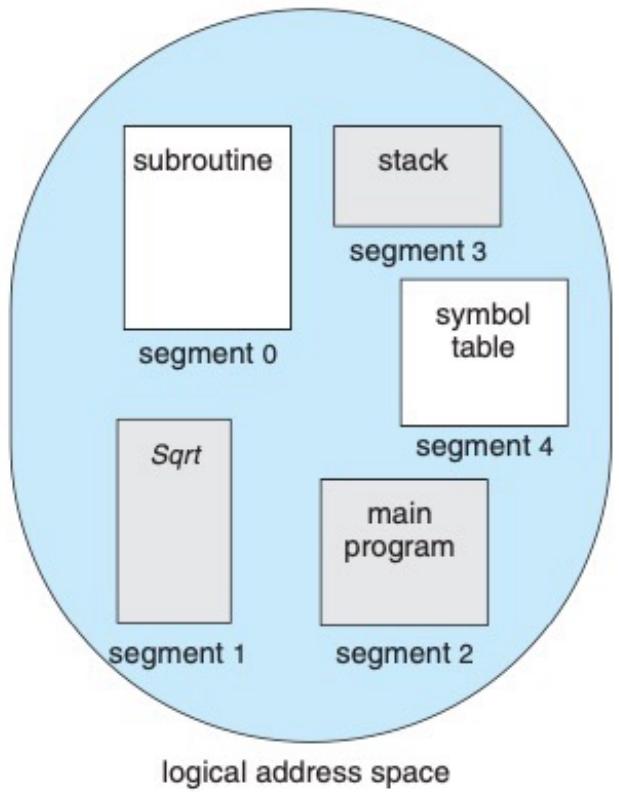
背景

- 允许程序大于物理内存空间，允许多个用户使用，
实际内存对用户透明
- 将内存抽象成一个巨大的、统一的存储数组，**实现了逻辑内存与物理内存的分离**，简化编程任务
- 可以运行更多程序，增加CPU利用率和吞吐量
- 虚拟内存可以通过“请求调页”来实现





段页结合的虚拟内存



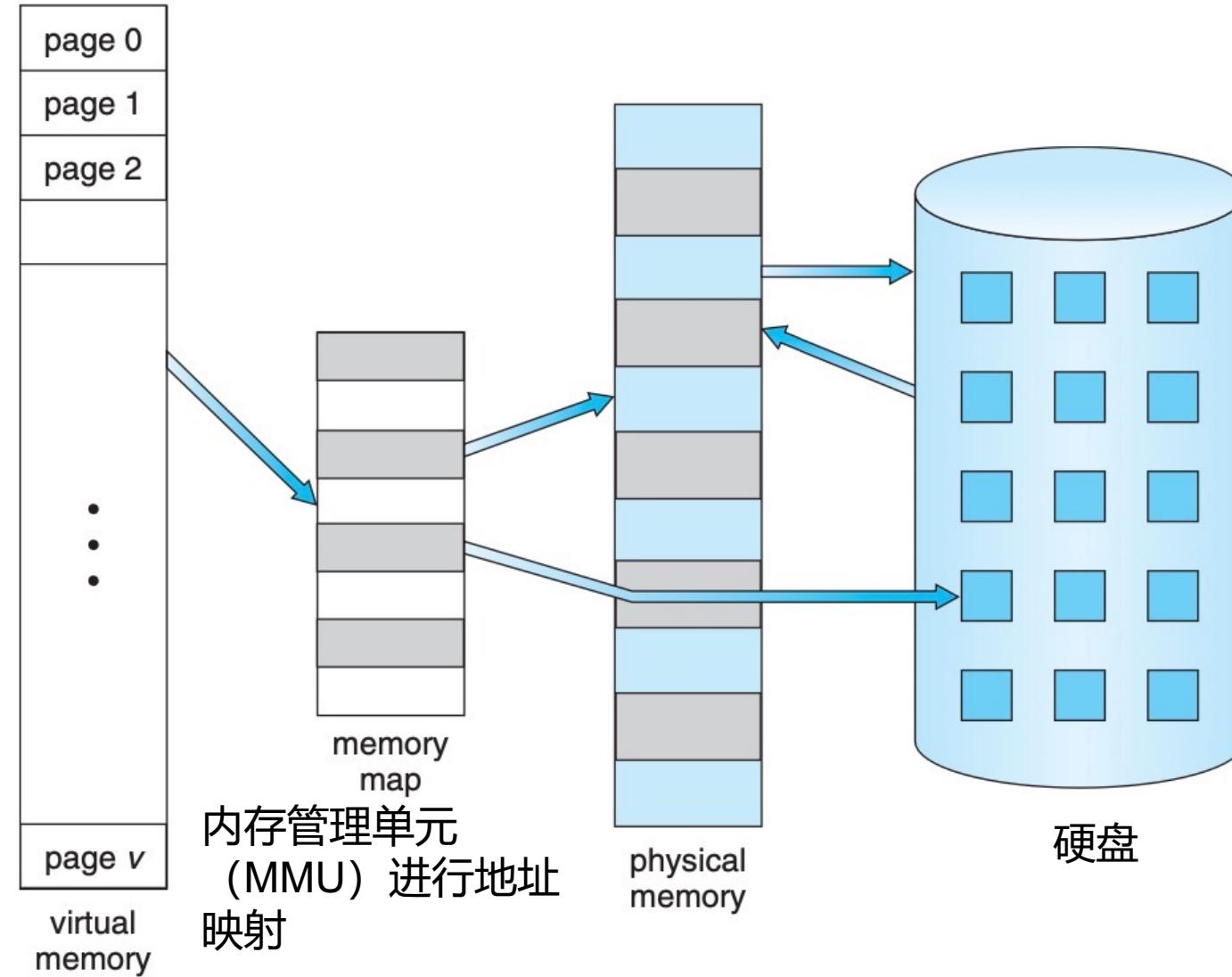
分段：从用户的角度将程序分段，存储到虚拟内存

分页：从物理内存来看，将虚拟内存的段分页





虚拟内存大于物理内存





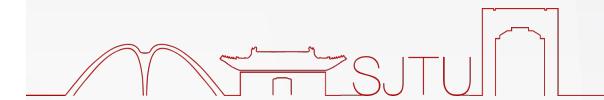
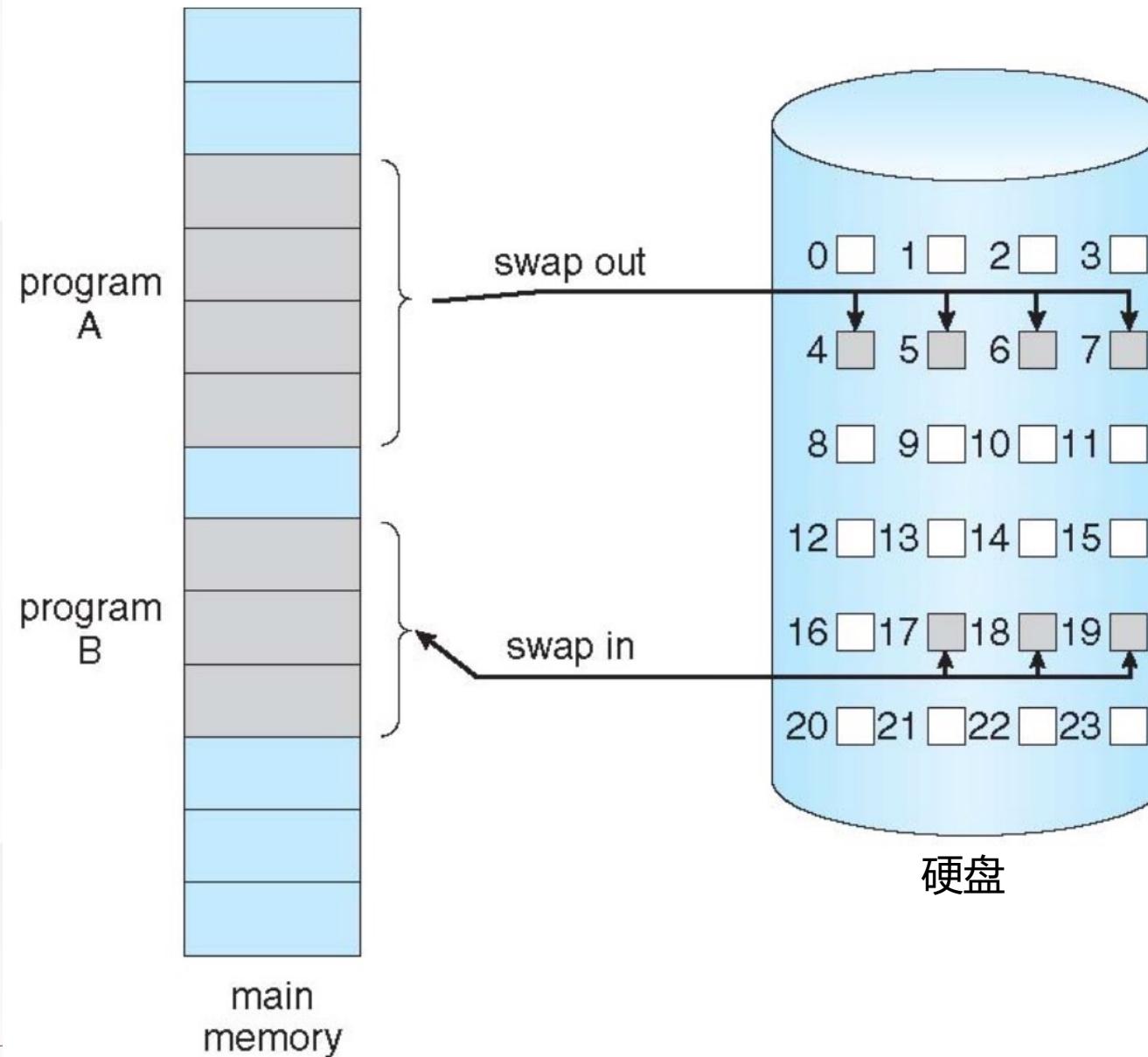
请求调页

- 当需要执行程序时，有两种方式加载程序：
 - 将整个程序加载到内存，空间浪费
 - 仅在需要时加载页面（请求调页）
- 当进程需要执行时，将所需页面交换到内存中，采用了惰性调用程序





请求调页





有效位与无效位



- 为了避免读入那些不使用的页，需要区分内存的页面和磁盘的页面，设置有效位与无效位
 - “有效” (valid, v)：页面合法，在内存中
 - “无效” (invalid, i)：页面无效（不在进程的逻辑地址空间中），或有效但仅在磁盘中
- 若访问未调入内存的页面，即访问无效页面，发生缺页错误 (page fault)

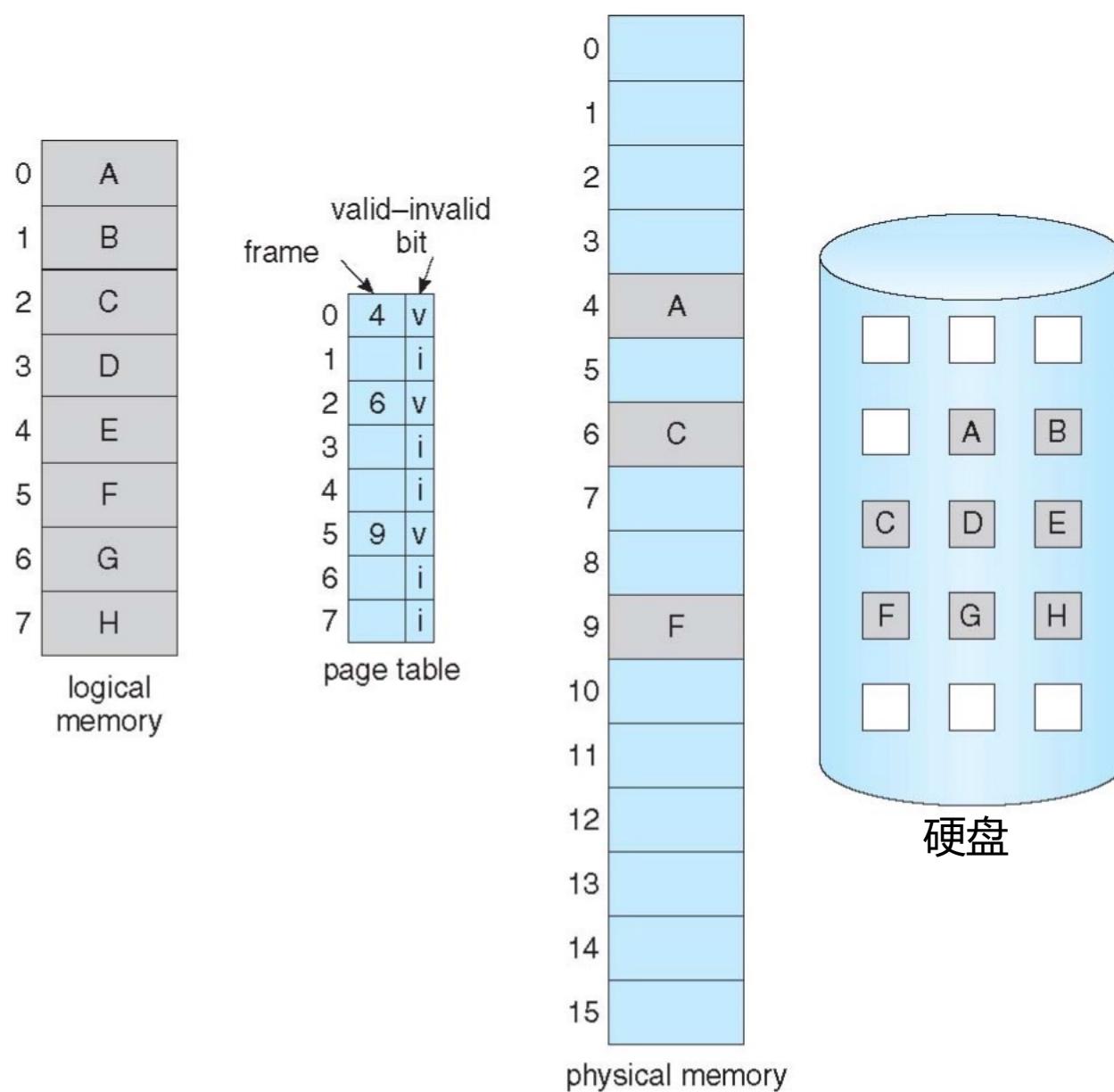
Frame #	valid-invalid bit
	v
	v
	v
	v
....	i
	i
	i

page table





部分页面不在内存的页表





缺页错误处理

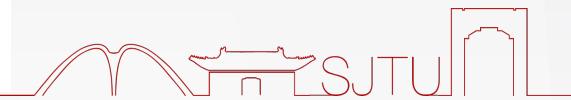
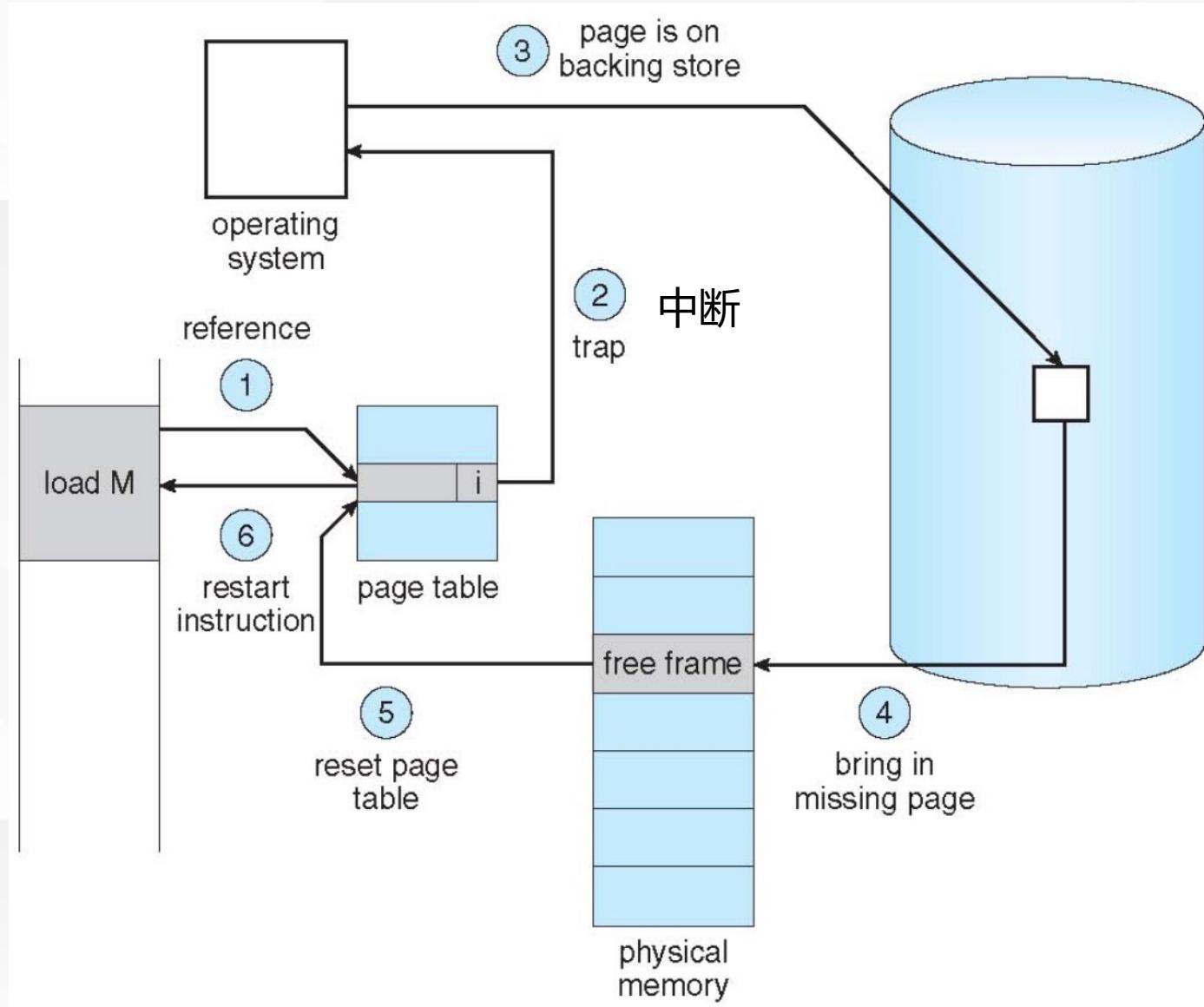


- 检查这个进程的内部表（通常与PCB一起保存），以确定该引用是有效的还是无效的内存访问
- 如果引用无效，那么终止进程。如果引用有效但是尚未调入页面，那么现在就应调入，**发出中断！进行缺页错误处理程序**
- 找到一个空闲帧(例如，从空闲帧链表上得到一个)
- 调度一个磁盘操作，以将所需页面读到刚分配的帧
- 当磁盘读取完成时，修改进程的内部表和页表v，以指示该页现在处于内存中
- 重新启动被陷阱中断的指令。该进程现在能访问所需的页面，就好像它总是在内存中





缺页错误处理

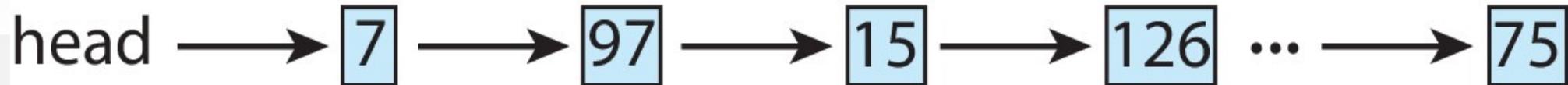




空闲帧列表



- 发生页面错误时，操作系统必须将所需页面从硬盘带入主存



- 大多数操作系统都维护一个空闲帧列表，即用于满足此类请求的可用帧池
- 操作系统通常使用一种称为“按需零填充”的技术来分配空闲帧，即帧的内容在分配之前被清零
- 当系统启动时，所有可用内存都放在空闲帧列表中





请求调页的性能



- 请求调页可以显著影响计算机系统的性能
- 计算请求调页内存的有效访问时间
- 设 p 为缺页错误的概率， m 为内存访问时间，则有效访问时间=
$$(1-p)xm + px\text{缺页错误时间}$$
- 缺页错误时间与缺页错误导致的行为有关：
 - 中断
 - 保存用户寄存器与进程状态
 - 检查页面引用是否合法、确定页面的磁盘位置
 - 从磁盘读入页面到空闲帧
 - ...





请求调页的性能



- 总的来说，缺页错误的处理时间有三个主要组成部分
 - 处理缺页错误中断
 - 读入页面
 - 重新启动进程
- 平均处理时间达到8ms，而内存访问时间约200ns，则有效内存访问时间=
$$(1-p)x200 + px8000000$$
- 有效访问时间与缺页错误率成正比





页面置换

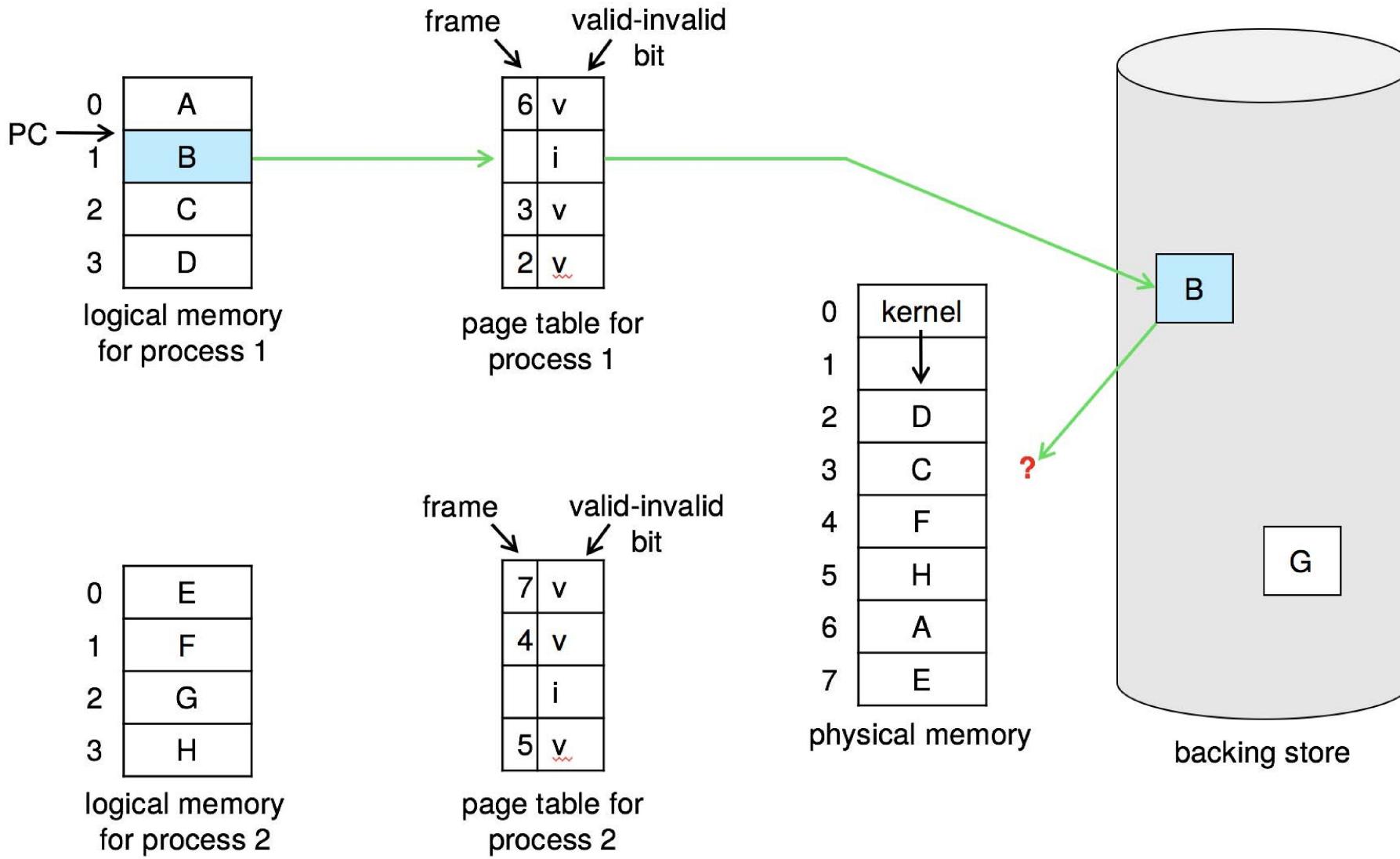


- 如果没有空闲帧可以放置了，该怎么办？
- 有几种选择
 - 将进程终止，显然不合理，请求调页是为了提升吞吐量，如果为此终止进程，则违背初衷
 - 换出一个进程---页面置换
 - 页面替换完成了逻辑内存和物理内存之间的分离
 - 可以在较小的物理内存上提供较大的虚拟内存



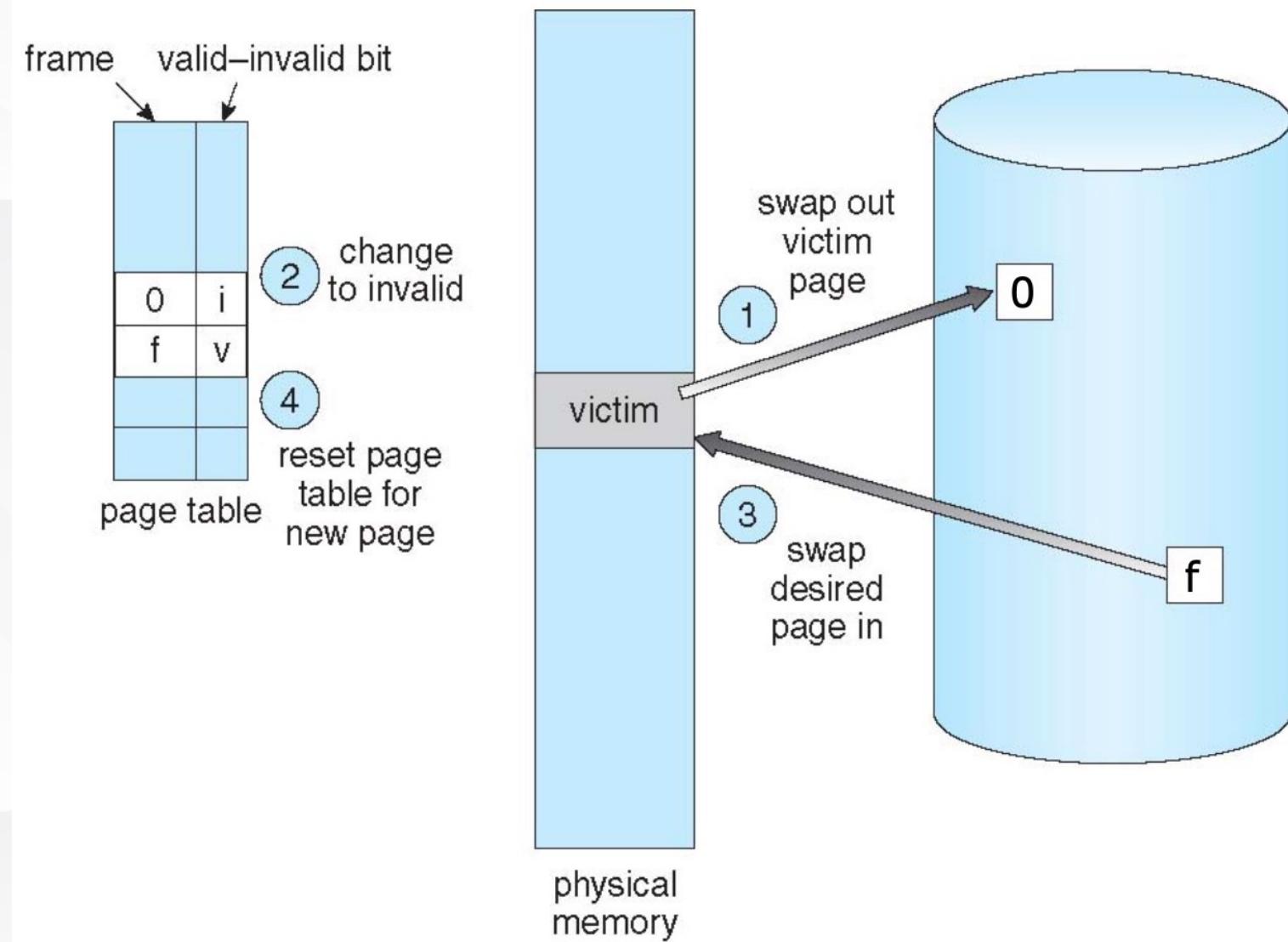


页面置换





页面置换





基本页面置换



- 修改缺页错误处理程序，以包括页面置换
 - 找到所需页面的磁盘位置
 - 找到一个空闲帧
 - 如果有空闲帧，则使用它
 - 如果没有空闲帧，则使用页面置换算法来选择一个**牺牲帧**
 - 将牺牲帧的内容写到磁盘，修改对应的页表和帧表
 - 将所需页面读入空闲帧，修改对应的页表和帧表
 - 从发生缺页错误位置，继续用户进程
- 注意，若出现页面置换，则需要两个页面传输（一个调入、一个调出），增加了访问时间！





基本页面置换



- 采用修改位（或脏位）可以减小访问开销
 - 当页面内的任何字节被写入时，它的页面修改位会由硬件来设置
 - 只有当修改位被设置，才需要将内存页面写回磁盘
 - 降低了一半的I/O时间





页面置换算法



- 有多种页面置换算法，如何选择？
 - 通常采用最小缺页错误率的算法
- 为了评估一个算法，针对特定内存引用串，运行某个置换算法，并计算缺页错误的数量
 - 内存引用的串称为引用串，可以人工生成（通过随机数生成器），或跟踪一个给定系统并记录内存引用的地址
 - 为了减少数据量，只需考虑页码，而非完整地址
- 我们在后续使用的引用串为：7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1





页面置换算法

- FIFO页面置换
- 最优页面置换
- LRU页面置换

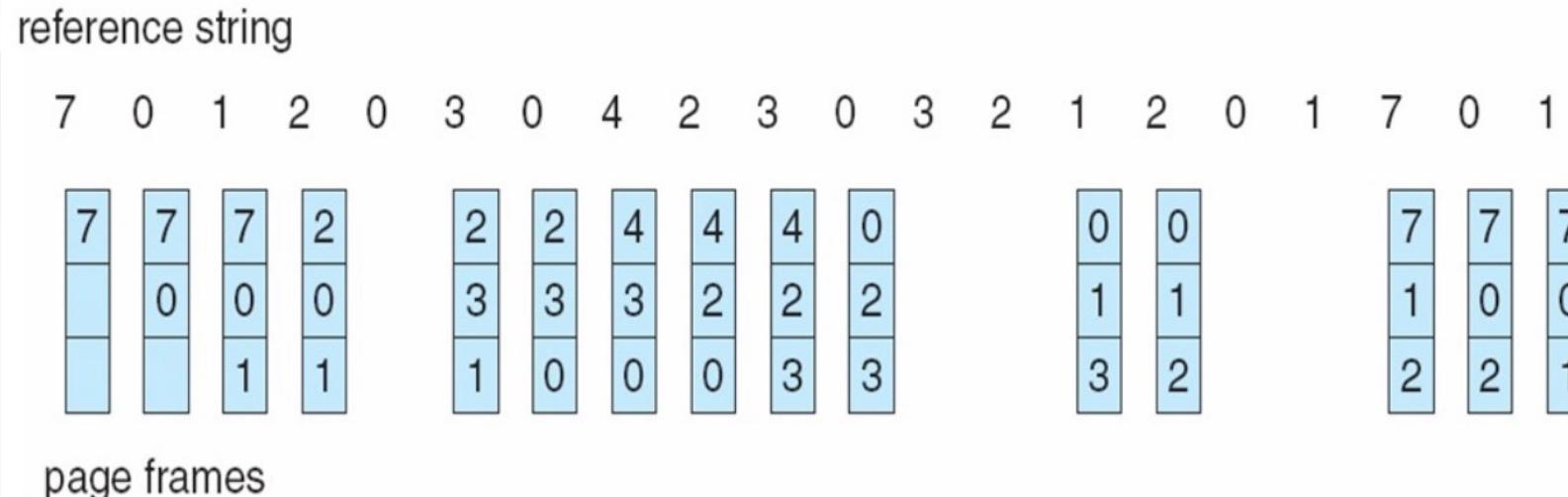




FIFO页面置换



- 为每个页面记录了调到内存的时间，最旧的页面将被换出



- 缺页错误发生15次
- 快速思考，下列引用串：0 1 2 3 0 1 2 3 0 1 2 3

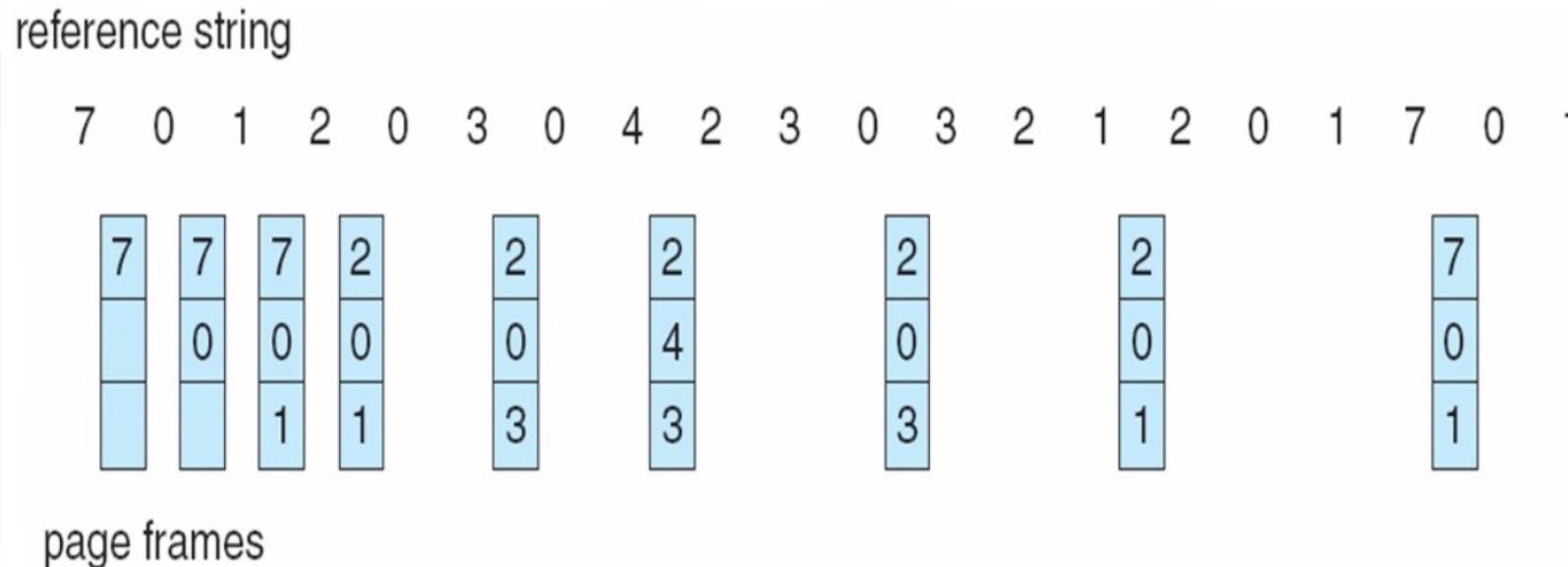




最优页面置换



- 置换最长时间不会使用的页面



- 缺页错误发生9次
- 但很难知道未来的信息，所以常用于比较研究，该方案作为理论最优解





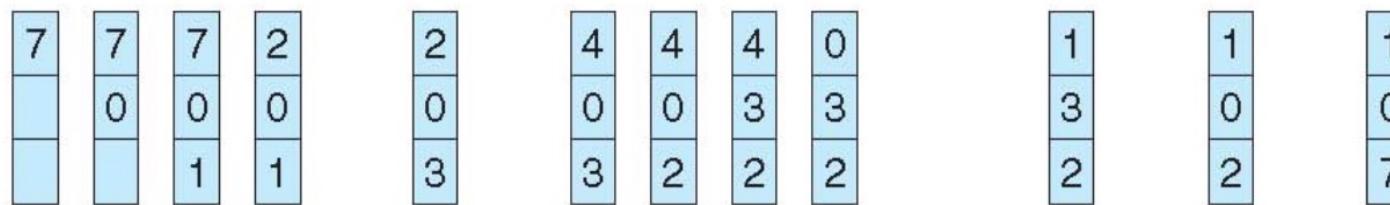
最近最少使用 (LRU) 页面置换



- 使用过去的信息，而非未来的信息
- 置换最长时间没有使用的页
- 将每个页面与它的上次使用的时间关联起

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

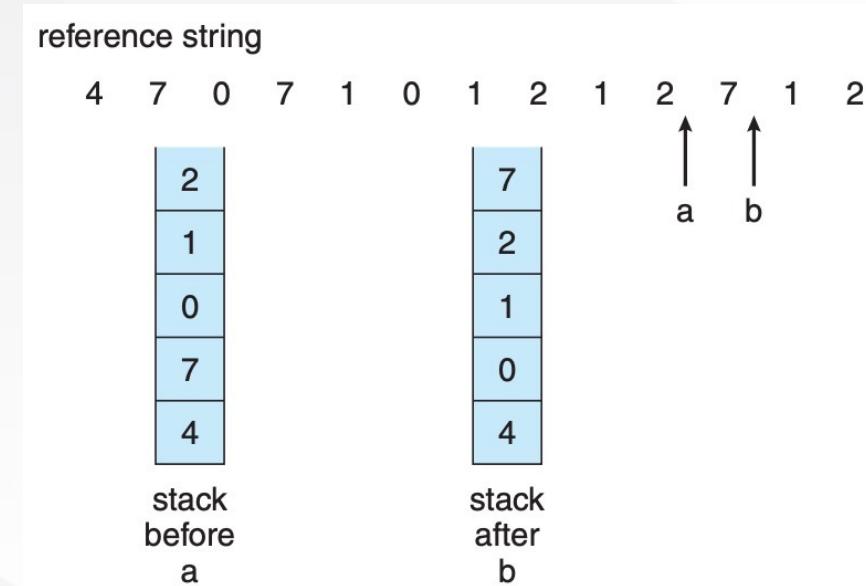
- 缺页错误发生12次，优于FIFO，比最优页面替换稍差
- 不错的策略，主要问题是实现LRU页面置换





最近最少使用 (LRU) 页面置换的实现

- 需要确定由上次使用时间定义的帧的顺序
 - 计数器
 - 为每个页表条目关联一个使用时间域，并为CPU添加逻辑时钟或计数器
 - 每次内存引用则增加时钟
 - 每次置换前，搜索页表，找到具有最小时间的页面进行替换
 - 堆栈
 - 每当页面被引用时，就从堆栈中移除并放在顶部





近似LRU页面置换



- 很少有计算机系统能提供足够的硬件来支持真正的LRU页面置换算法
- 通过引用位 (reference bit) 的形式对页面置换提供一定的支持
 - 每个条目有一个引用位，初始化为0
 - 被引用的页面的引用位被设置为1
- 近似LRU页面置换算法
 - 额外引用位算法
 - 第二次机会算法





额外引用位算法



- 为每个页面保留一个8位的字节
- 8位移位寄存器包含最近8个时钟周期的页面使用情况，每个时钟周期向右移动1位，更新最左侧位
 - 如果移位寄存器包含00000000，则该页面在8个周期内没有被使用
 - 具有11000100 的移位寄存器值的页面比具有值为01110111 的页面更为“最近使用的”
- 如果将这些8位字节解释为无符号整数，那么具有最小数值量的寄存器所对应的页面可以被替换





第二次机会算法

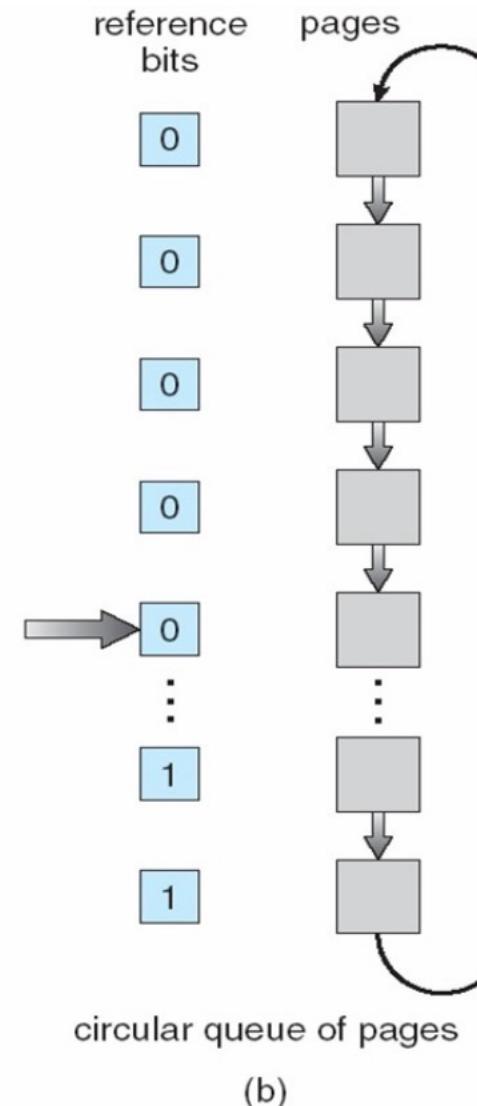
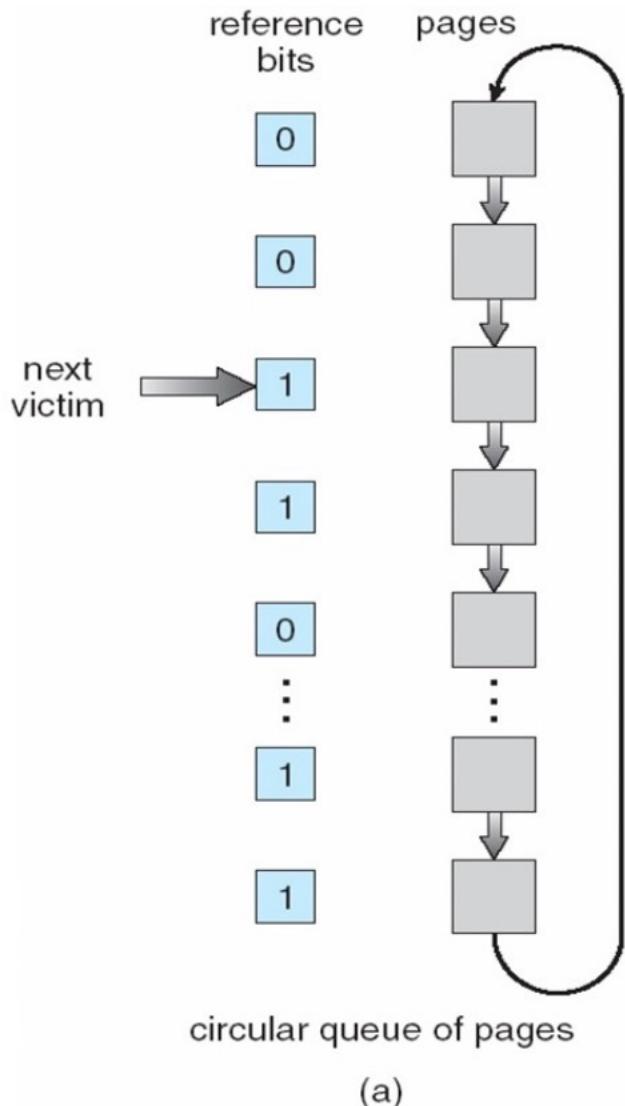


- 第二次机会置换的基本算法是一种FIFO置换算法+引用位辅助
- 当选择了一个页面，则检查其引用位
 - 如果引用位=0，则直接替换该页面
 - 如果引用位=1：
 - 将引用位置为0，给它第二次机会
 - 选择下一个页面





第二次机会算法



如果所有引用位都是1，就退化成FIFO置换算法





增强型第二次机会算法



- 通过将引用位和修改位作为有序对，改进第二次机会算法
 - (0, 0)最近没有使用且没有修改的页面，最佳的页面置换
 - (0, 1)最近没有使用但修改过的页面，不太好的置换，因为在置换之前需要将页面写出
 - (1, 0)最近使用过但没有修改的页面，可能很快再次使用
 - (1, 1)最近使用过且修改过，可能很快再次使用，并且在置换之前需要将页面写出到磁盘
- 给已修改页面赋予更高级别，降低I/O频率





基于计数的页面置换



- 可以为每个页面的引用次数保存一个计数器
 - 最不经常使用页面置换算法
 - 将最小引用次数的页面置换
 - 最经常使用页面置换算法
 - 基于以下论点：具有最小计数页面可能是刚刚被引入而尚未被使用的





帧分配



- 在各进程间，如何分配固定数量的可用内存？
 - 假如有93个空闲帧和2个进程，那么每个进程各有多少帧？
- 帧分配策略受到多方面的限制，其中之一是：所分配的帧有最小数量要求
 - 原因是考虑性能损失
 - 若分配的帧数量过少，缺页错误率增加，有效访问内存时间会相应增加





帧分配算法



- 平均分配
 - 在n个进程中分配m帧，给每个进程一个平均值，即 m/n 帧
 - 有些进程不需要这么多帧，造成资源浪费
- 比例分配
 - 根据每个进程大小分配帧
 - 但比例分配没有考虑进程的优先级
- 基于优先级的分配策略
 - 优先级越高，分配帧越多，使其执行速度加快，提升整体性能





帧分配算法 全局分配与局部分配



- 全局置换
 - 允许一个进程从所有帧的集合中选择一个置换帧
 - 可以从另一个进程那里获取帧
 - 灵活度更高，增加了分配给它的帧数
 - 可能导致系统抖动
- 局部置换
 - 每个进程只从它自己分配的帧中进行选择

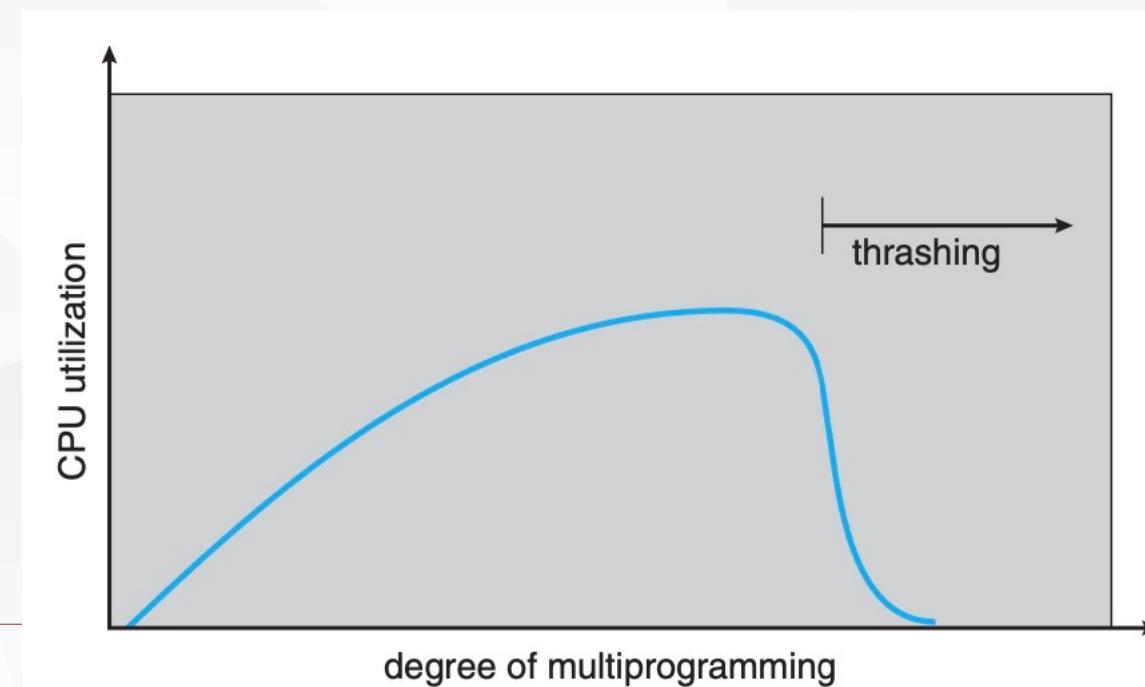




系统抖动



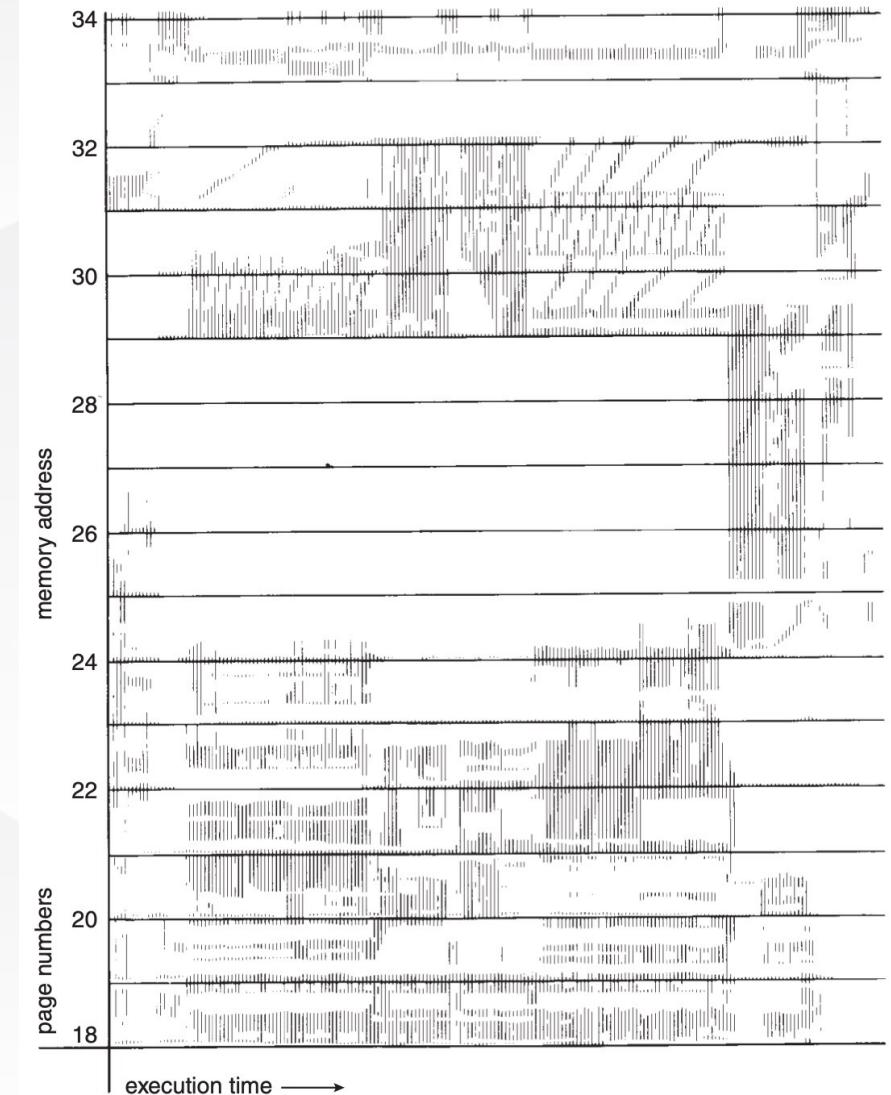
- 如果一个进程的调页时间多于它的执行时间，那么这个进程处于抖动
- 系统抖动将造成严重的性能问题
- 系统抖动的原因：CPU调度程序看到CPU利用率降低，增加多道程序，需要获取更多帧，导致更多的抢占、缺页错误，因此CPU利用率进一步下降，产生抖动





系统抖动

- 为了提高CPU利用率并停止抖动，需要降低多道程序
 - 通过局部置换算法或优先级置换算法，不允许进程从其他进程中获取帧
- 为了防止抖动，还应该为进程提供足够多的帧数。那么应该如何知道进程所需的帧数？
 - 工作集策略，定义了进程执行的局部性模型
 - 如果为进程分配了足够的帧来适应当前的局部性，不会抖动





工作集模型



- 工作集：一个进程当前正在使用的逻辑页面集合，可用一个二元函数 $W(t, \Delta)$ 来表示
- t 是当前的执行时刻
- Δ 是工作集窗口，即一个定长的页面访问的时间窗口
- $W(t, \Delta) =$ 在当前时刻 t 之前的 Δ 时间窗口当中所有页面的集合（随着 t 变化）
- $|W(t, \Delta)|$ 表示工作集大小，即页面数量



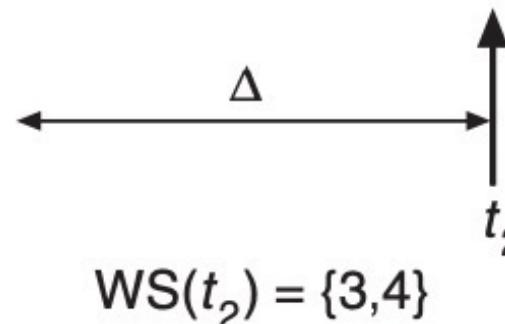
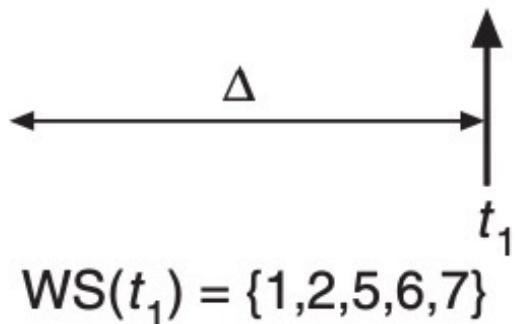
工作集模型



- $W(t_1, \Delta) = \{1, 2, 5, 6, 7\}$
- $W(t_2, \Delta) = \{3, 4\}$
- t_2 的局部性比 t_1 好

page reference table

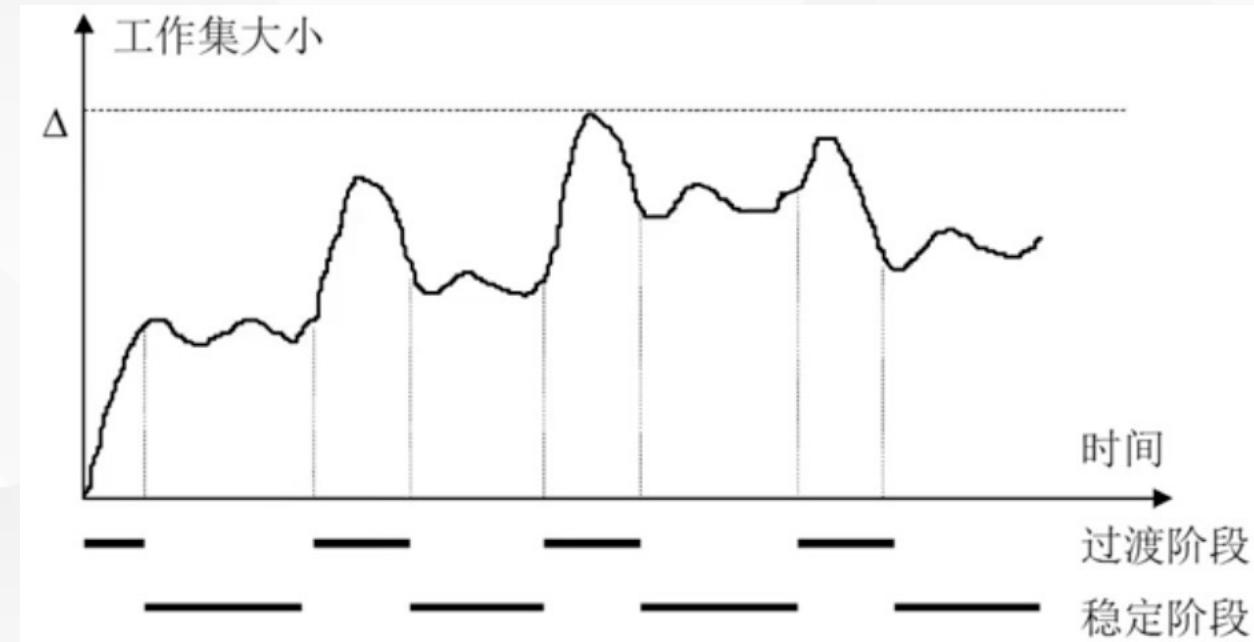
... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...





工作集模型

- 工作集大小的变化：进程开始执行后，随着访问新页面逐步建立较稳定的工作集。当内存访问的局部性位置大致稳定时，工作集大小也较为稳定；局部性区域位置改变时，工作集快速扩张和收缩，过渡到下一个稳定值





课题习题



- 考虑下面的页面引用串: 7, 2, 3, 4, 2, 1, 5, 3, 4, 7, 7, 6, 1, 0
- 假设采用4个帧的请求调页, 以下置换算法会发生多少次缺页错误?
 - LRU置换
 - FIFO置换
 - 最优置换