

第9章 虚拟内存



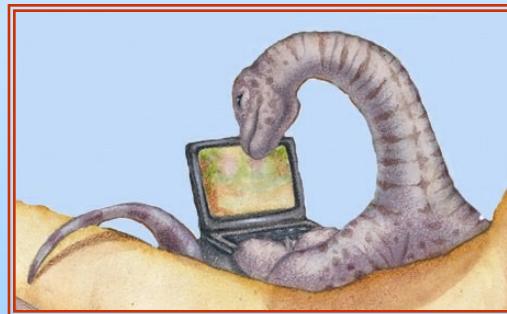


内容

- 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

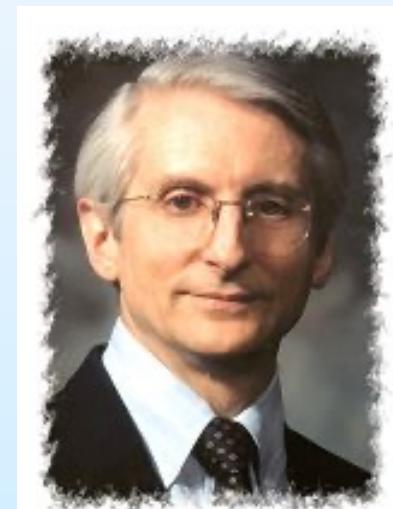




局部性原理

■ 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

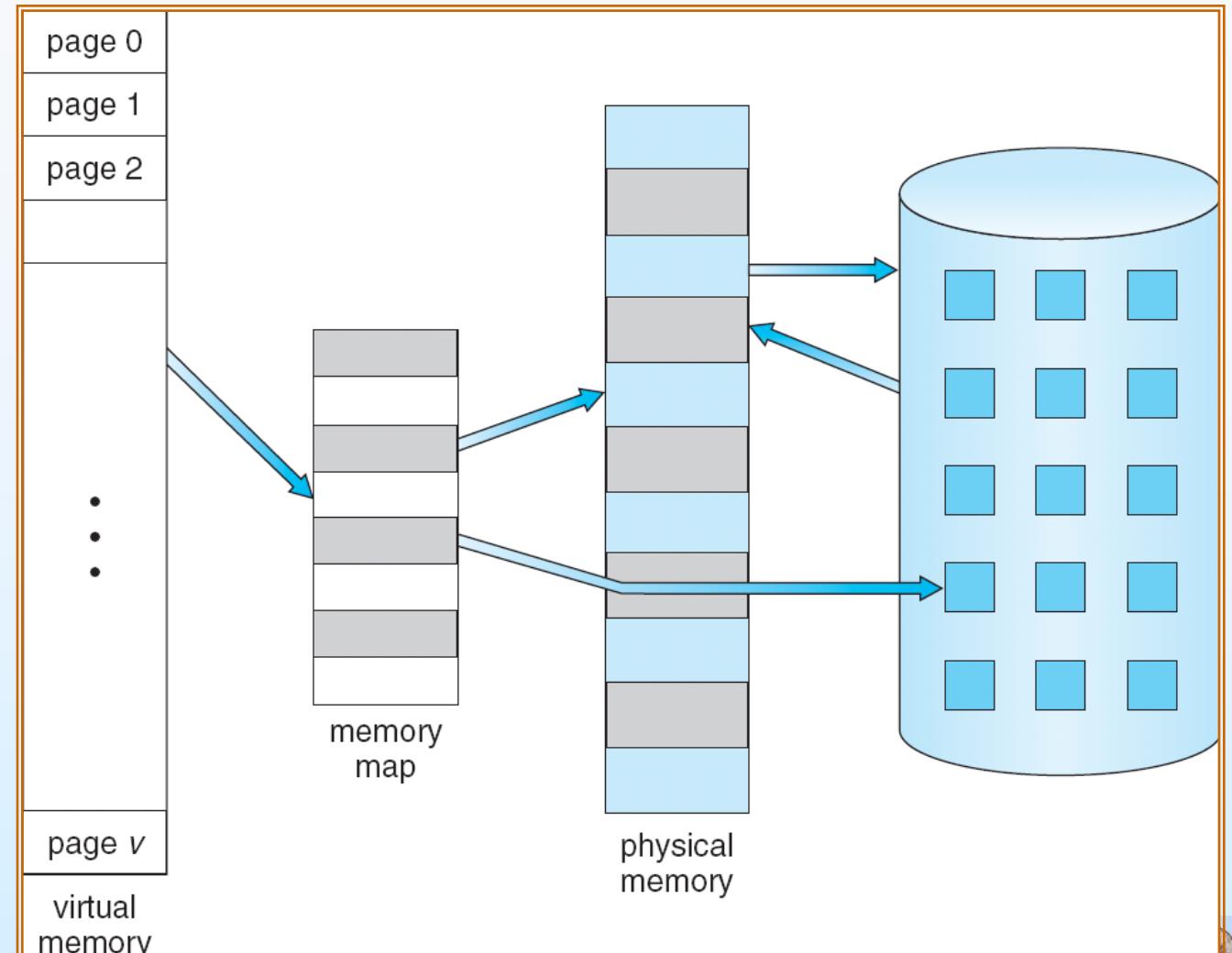




虚拟内存大于物理内存

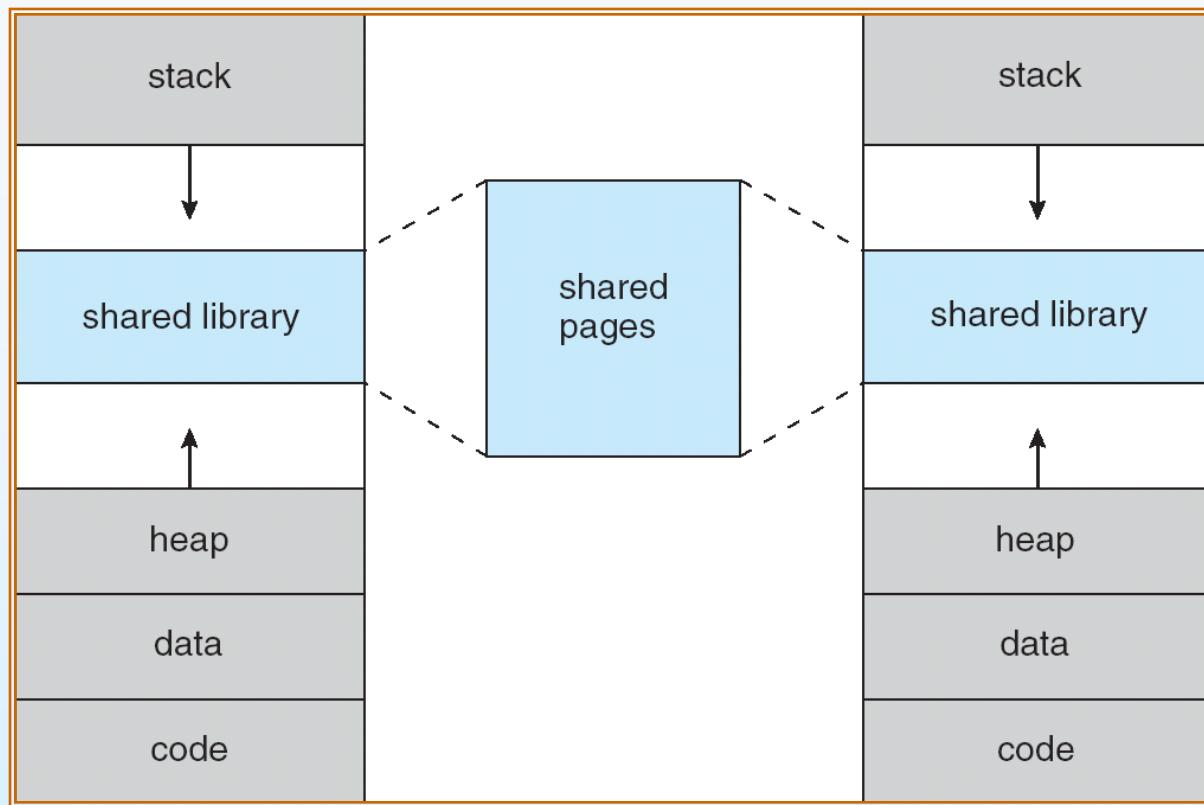
虚拟存储器的大小由**2**个因素决定

- 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）

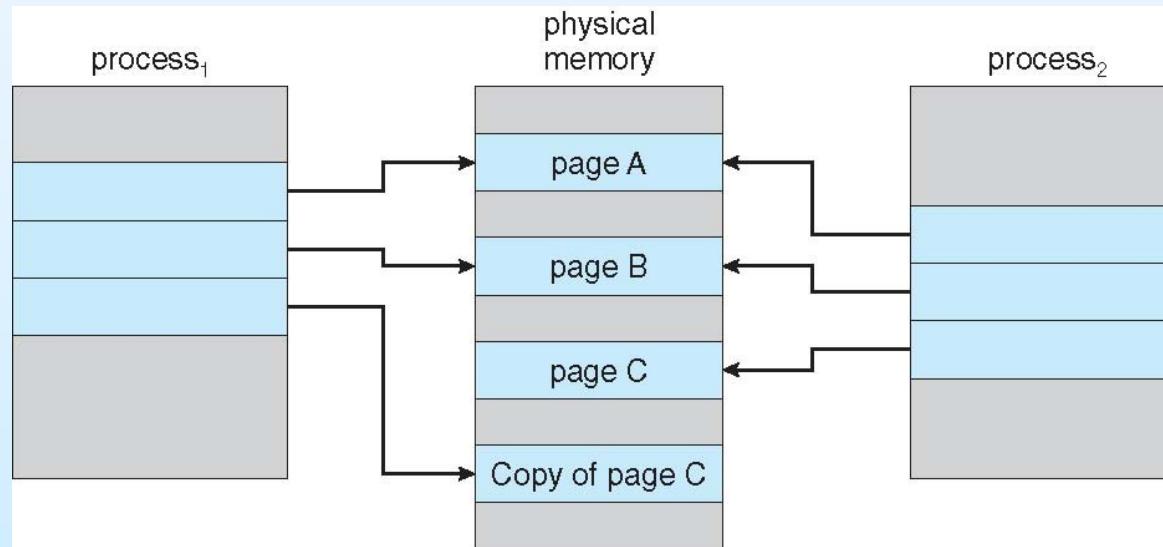
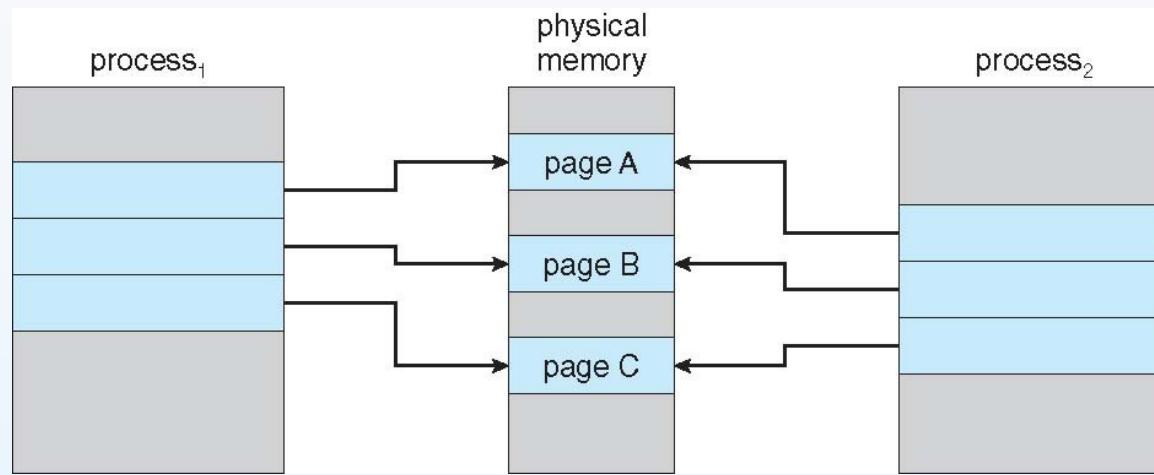
- 写时复制允许父进程和子进程在初始化时共享页面
 - 如果其中一个进程修改了一个共享页面，会产生副本
 - 更加高效
 - 应用在Windows XP, Linux等系统

- vfork: fork() 变形，不使用写时复制





写时复制例子





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

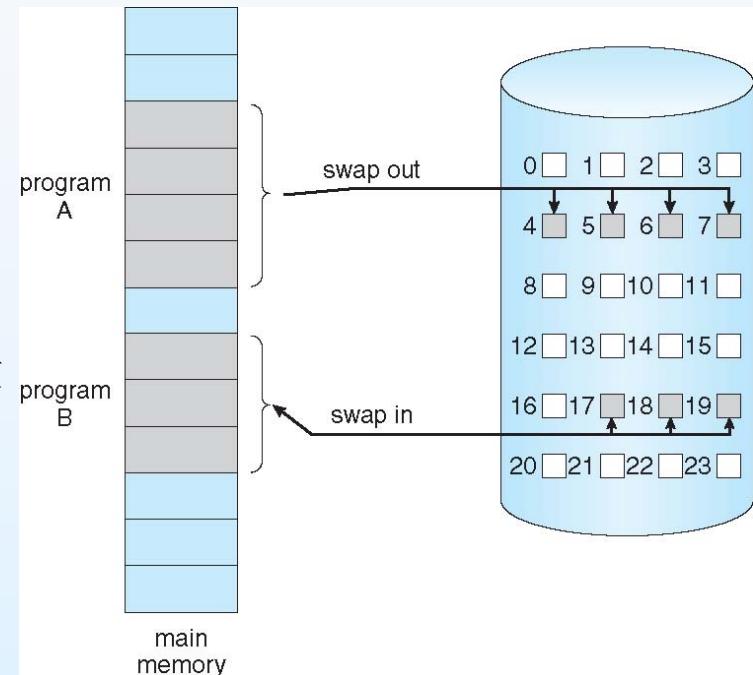
- 交换程序（**swapper**）对整个进程进行操作
- 调页程序（**pager**）只是对进程的单个页进行操作

■ 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都有一个有效- 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

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
:	
	0
	0

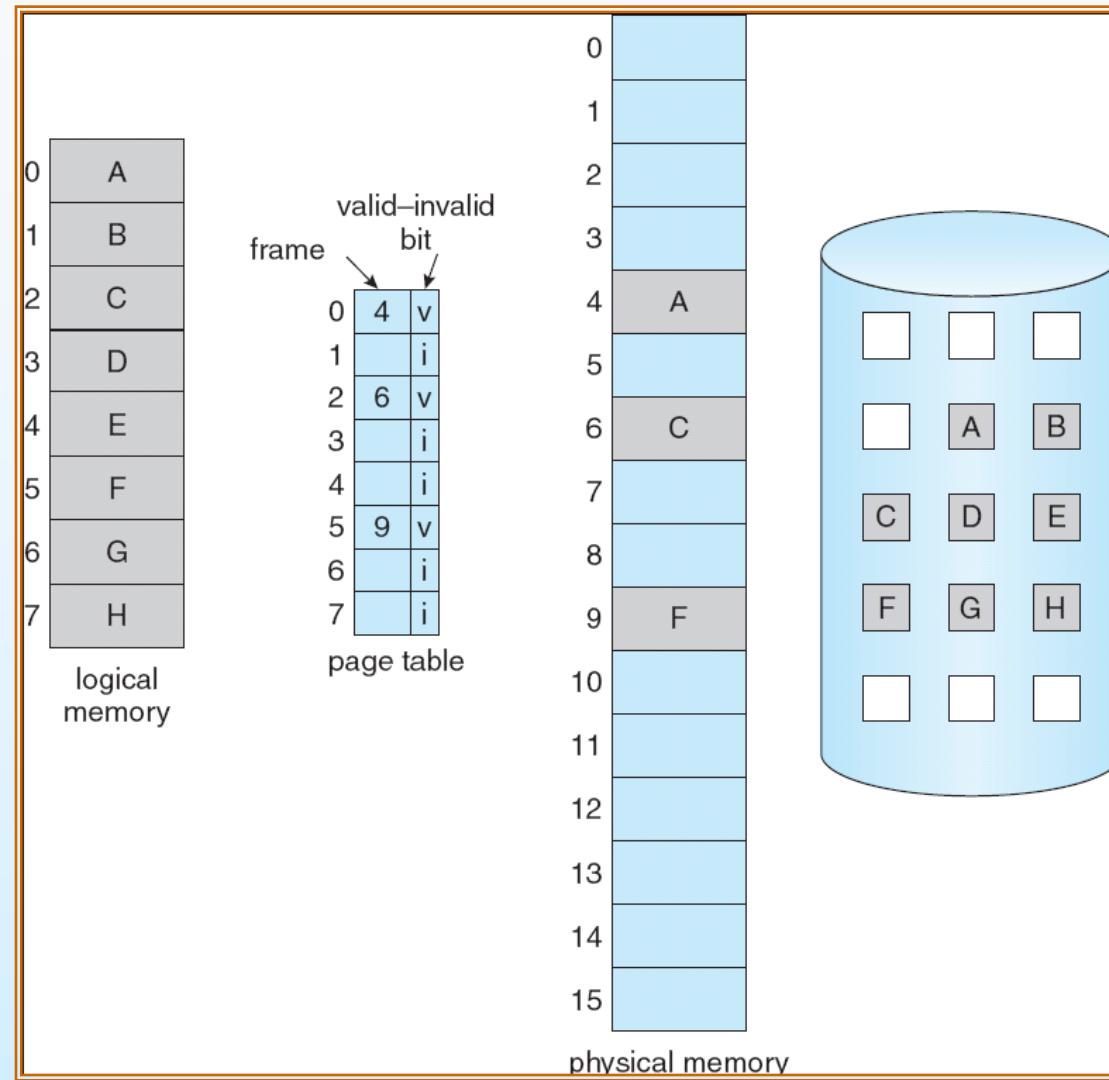
page table

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





有页不在内存的页表





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 \Rightarrow 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

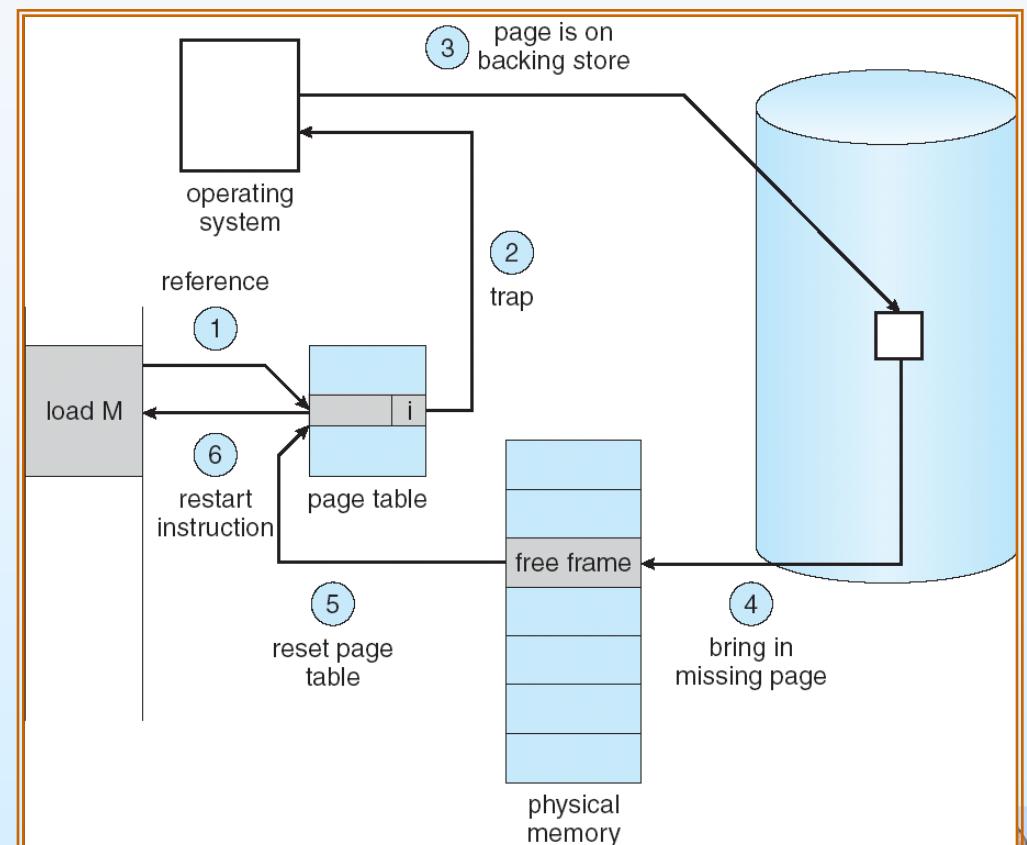
- 无效引用 \Rightarrow 终止
- 仅仅不在内存

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

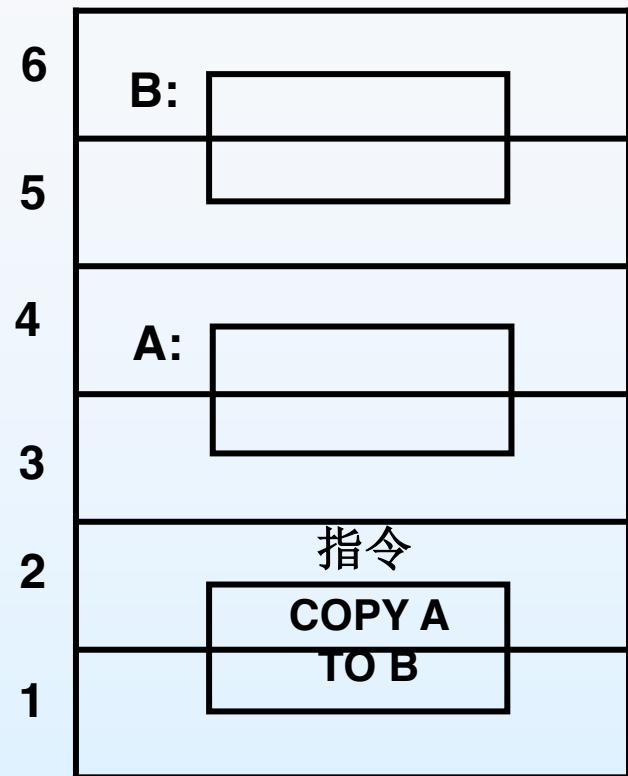
6. 重启指令：最近未使用





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: $0 \leq p \leq 1.0$

- 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)
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$
$$= (1 - p) \times 200 + p \times 8,000,000$$
$$= 200 + p \times 7,999,800$$
- 如果每1,000次访问中有一个页错误，那么
 $EAT = 8.2 \text{ us}$

这是导致计算机速度放慢40倍的影响因子！



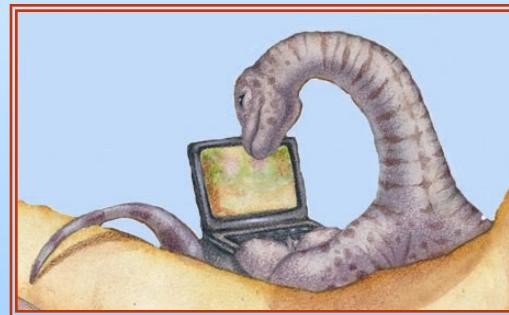


请求分页性能优化

- 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）

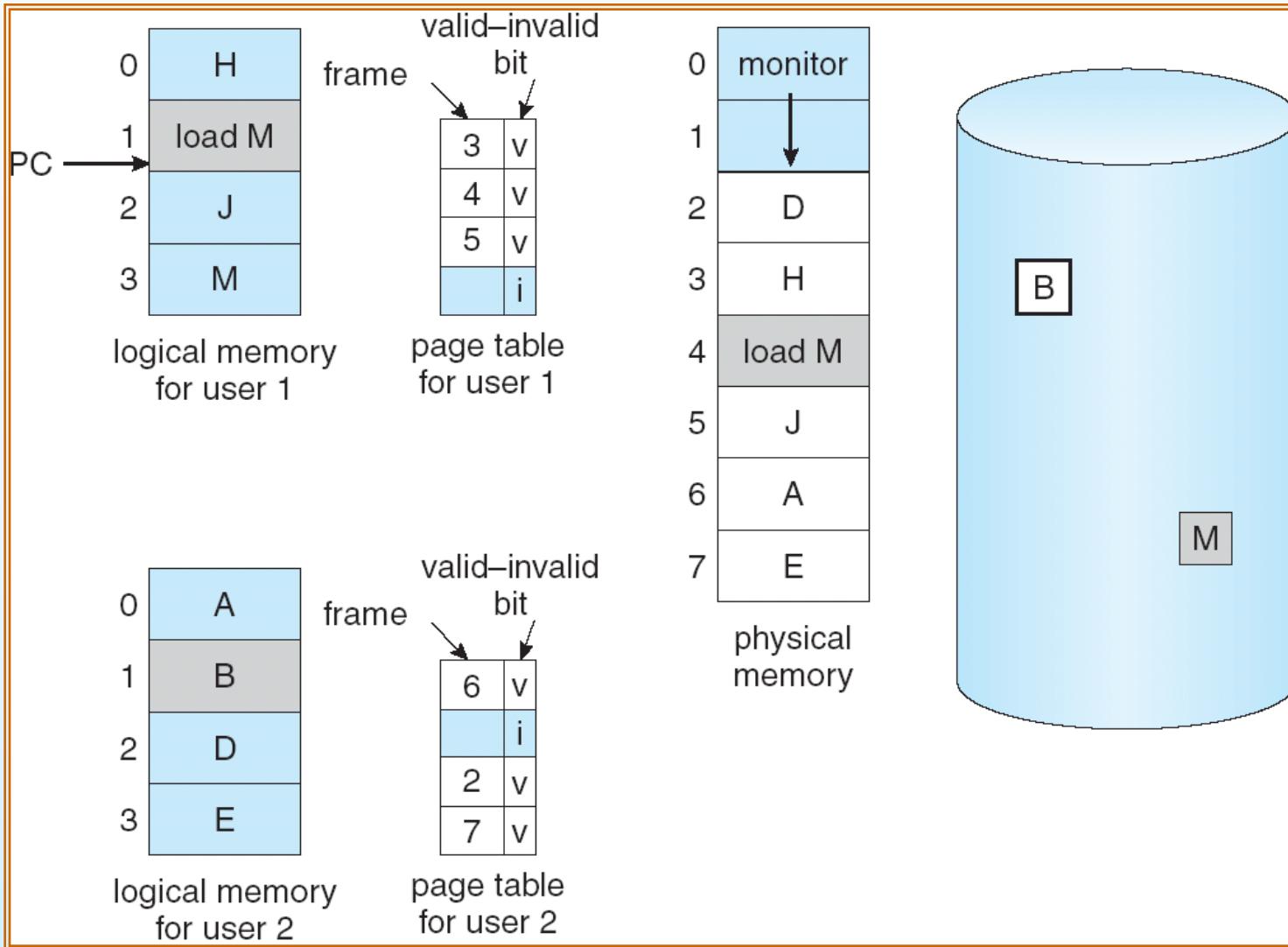


3、页面置换





需要页置换的情况





如果没有空闲页怎么办？

■ 解决方法：

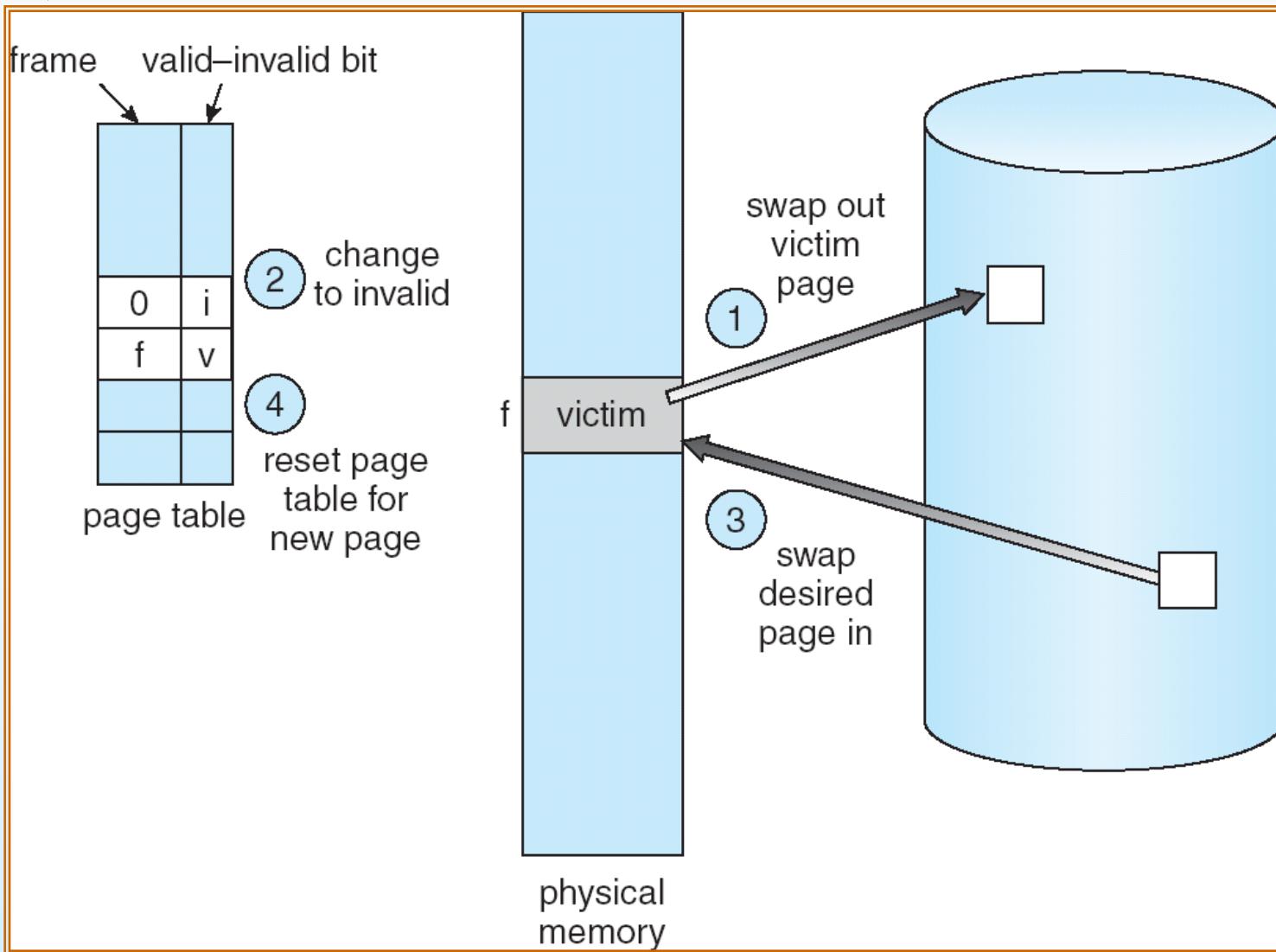
- 终止进程
- 交换进程
- 页面置换（page replacement），又称页置换、页淘汰

■ 页面置换

- 找到内存中没有使用的一些页，换出
- 同一个页可能会被装入内存多次



页置换





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

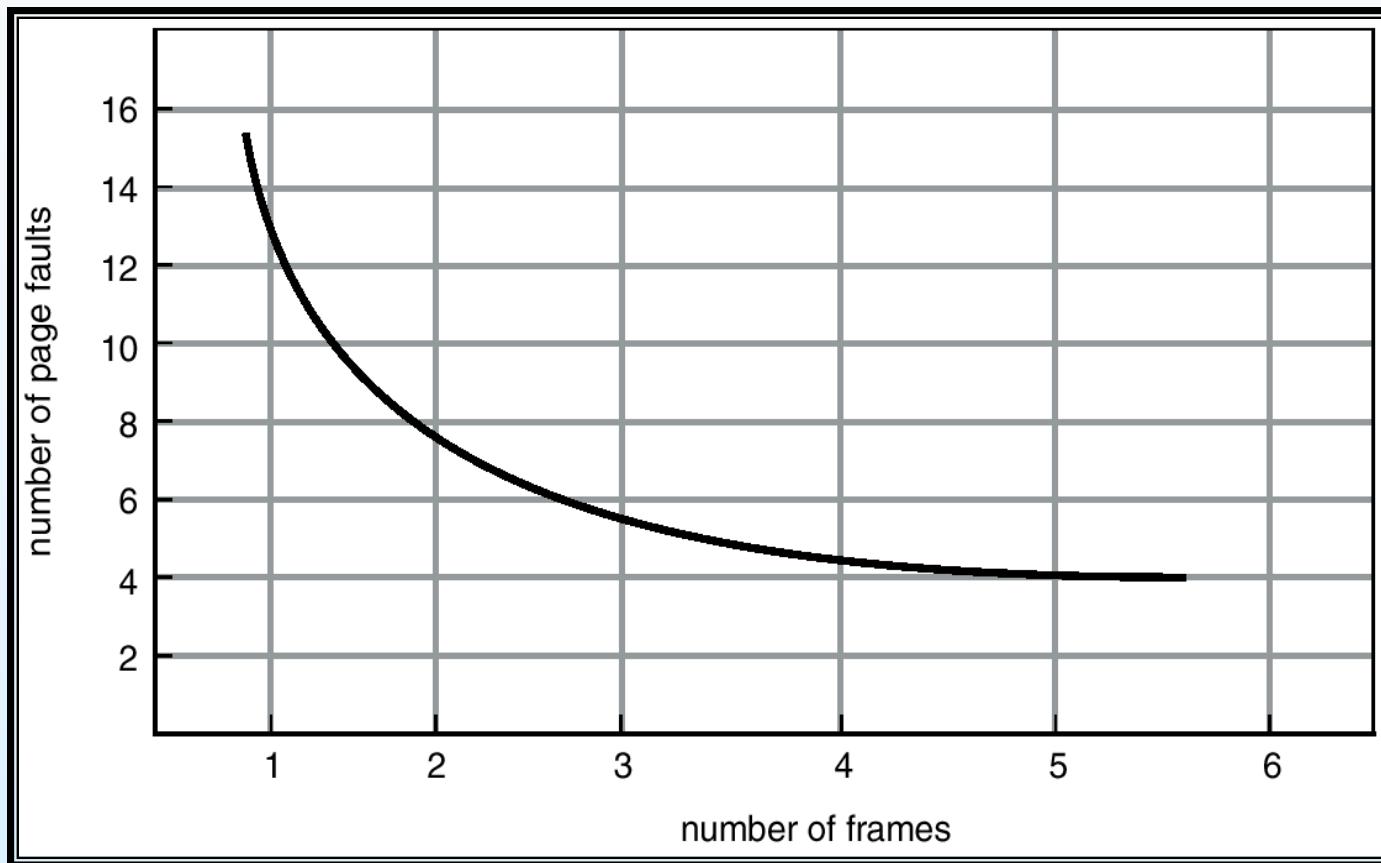
■ 页面置换算法

- 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





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



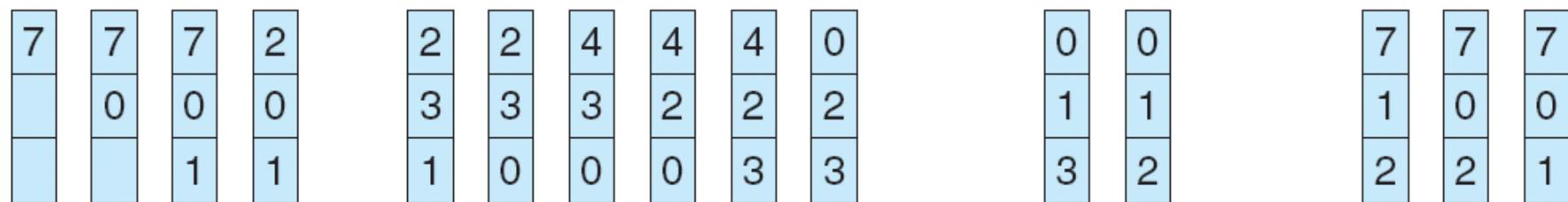


先进先出(FIFO)算法

- 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

reference string

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



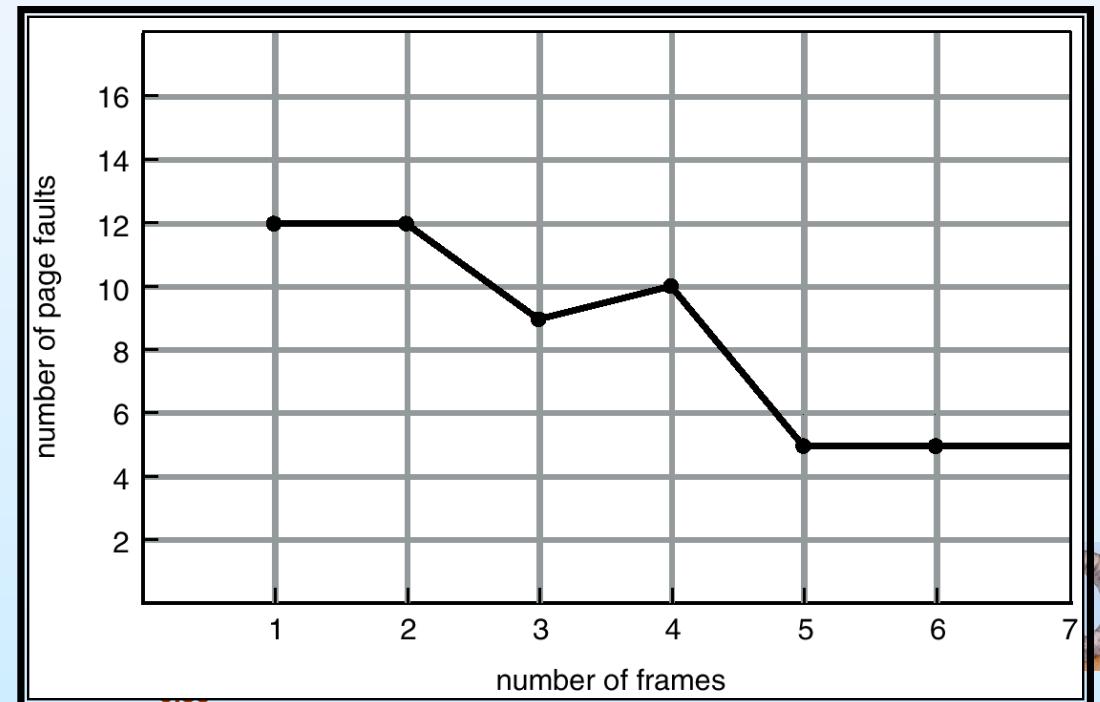
page frames





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 \Rightarrow 更多的缺页

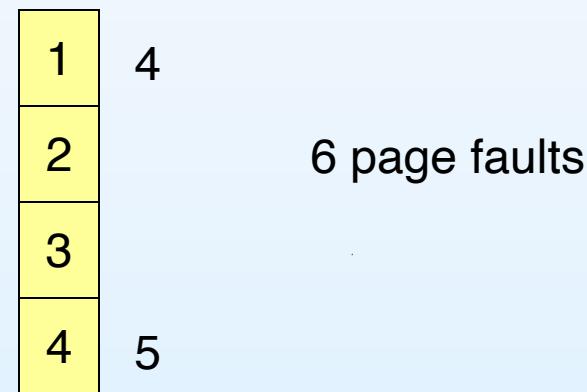




最优置换算法 (OPT)

- 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



- 怎样知道的？
- 作用：作为一种标准衡量其他算法的性能

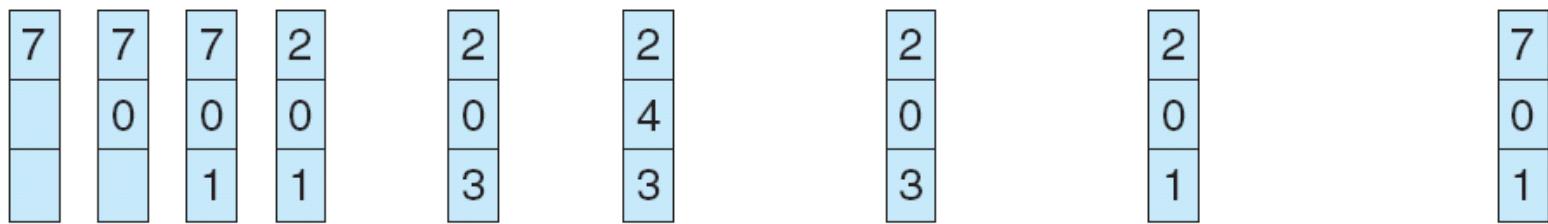




最优置换算法

reference string

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



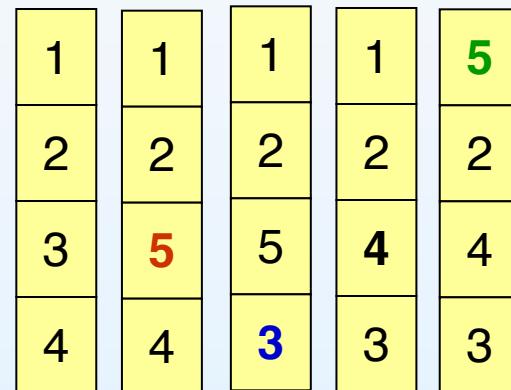
page frames





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

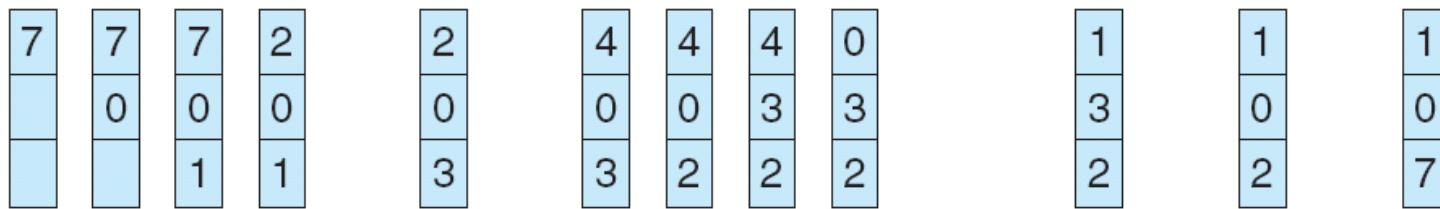




LRU 置换算法

reference string

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



page frames





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

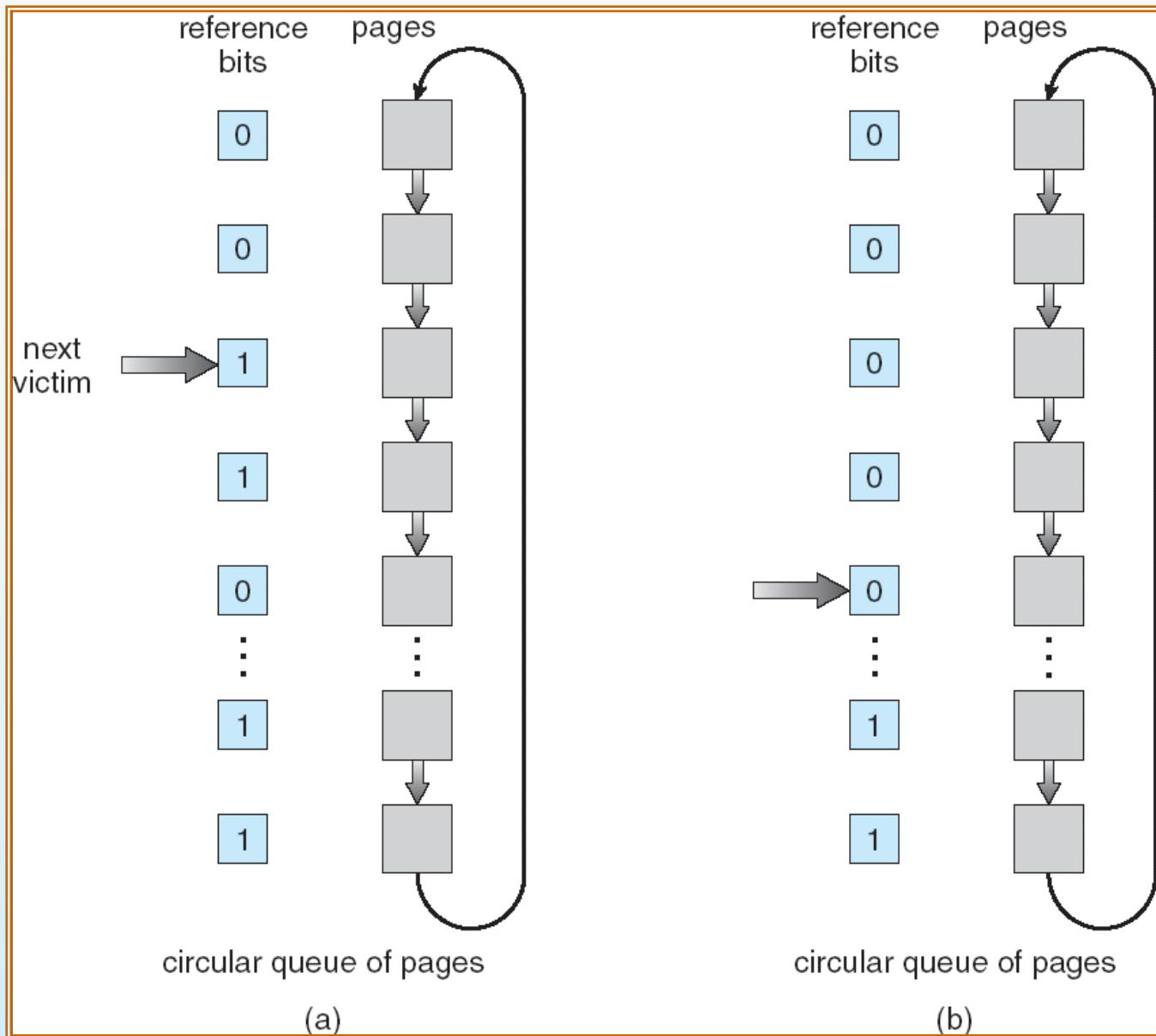
■ 二次机会算法（基本算法FIFO）

- 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）

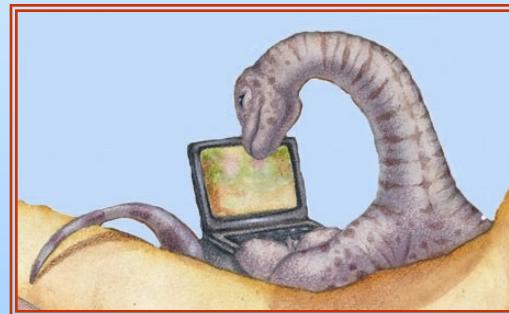




二次机会置换算法



4、页框分配





页框的分配

- 也称帧分配，就是研究如何为各个进程分配一定的空闲内存
- 如：现有93个空闲页框和2个进程，那么每个进程各得到多少页框？
- 页框分配会受到多方面的限制。例如，所分配的页框不能超过可用页框的数量，也必须分配至少最少的页框数，即每个进程所需要的最少的页框数。
- 分配至少最少的页框数的原因之一是性能
 - 随着分配给每个进程的页框数量的减少，缺页中断会增加，从而减慢进程的执行
 - 当指令完成之前出现缺页中断时，该指令必须重新执行，因此必须有足够的页框来容纳所有单个指令所引用的页





页框的分配

- 必须满足：每个进程所需要最少的页数
- 例子：IBM 370 – 6 处理 **SS MOVE** 指令：
 - 指令是 6 个字节，可能跨越 2 页
 - 2 页处理 from
 - 2 页处理 to
- 两个主要的分配策略。
 - 固定分配
 - 优先分配





固定分配

- 为每个进程分配固定数量的页框，也有两种分配方式：
- 平均分配 – 均分法
 - 例：如果有100个页框，和5个进程，则每个进程分给20个页
- 按比率分配 – 根据每个进程的大小来分配

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 4$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$





优先级分配

- 根据优先级而不是进程大小来使用比率分配策略
- 如果进程 P_i 产生一个缺页
 - 选择替换其中的一个页框
 - 从一个较低优先级的进程中选择一个页面来替换





全局置换和局部置换

■ 全局置换

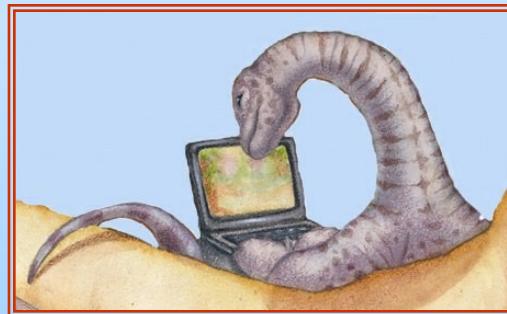
- 进程在所有的页中选择一个替换页面；一个进程可以从另一个进程中获得页面

■ 局部置换

- 每个进程只从属于它自己的分配页框中选择
- 采用局部置换，分配给每个进程的页框**数量不变**，采用全局置换，**可能增加所分配页框的数量**，因为可能从分配给其他进程的页框中选择一个置换
- 全局置换的问题，进程不能控制其缺页率，局部置换没有这个问题。但局部置换不能使用其他进程不常用的内存。
- 全局置换有更好的系统吞吐量，更为常用。



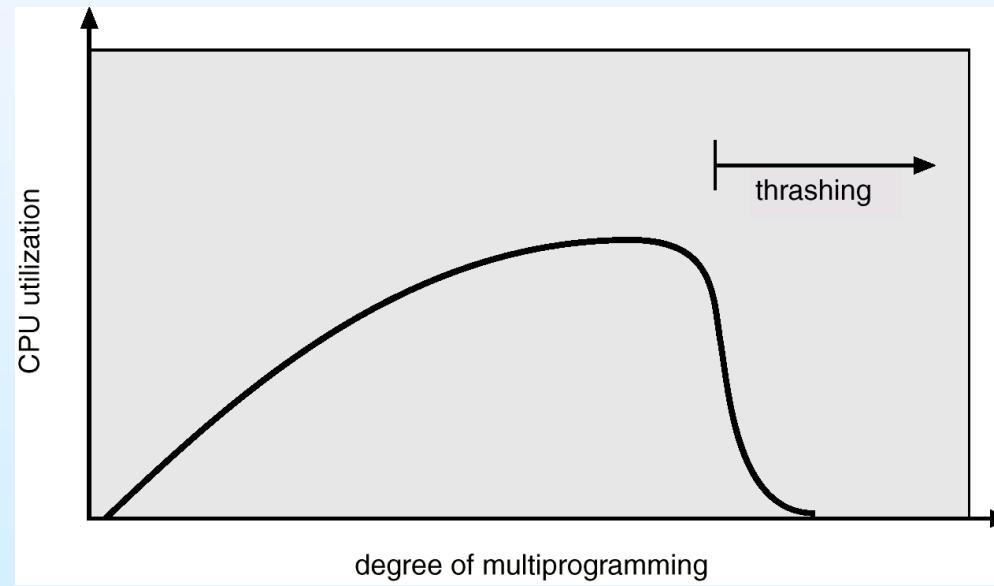
5、颠簸





颠簸Thrashing

- 如果一个进程没有足够的页，那么缺页率将很高，这将导致：
 - CPU利用率低下.
 - 操作系统认为需要增加多道程序设计的道数
 - 系统中将加入一个新的进程
- 颠簸 \equiv 一个进程的页面经常换入换出



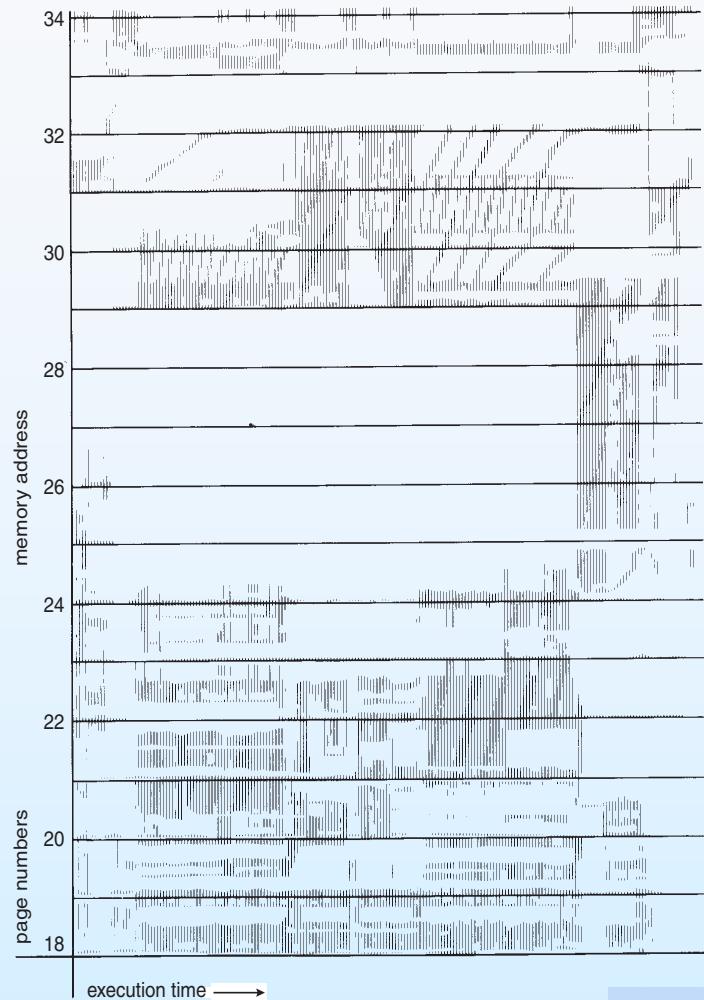
引发这种现象的原因：系统内存不足，页面置换算法不合理





局部置换算法 local replacement algorithm

- 通过局部置换算法可以限制系统颠簸
- 如果一个进程开始颠簸，那么它不能置换其他进程的页框
- 局部模型（Locality model）
 - 进程从一个局部移到另一个局部
 - 局部可能重叠
- 为什么颠簸会发生
 $\Sigma \text{ size of locality} > \text{total memory size}$





工作集模型(working-set model)

- $\Delta \equiv$ 工作集窗口 \equiv 固定数目的页的引用
例如: 10,000 个引用
- $WSS_i (P_i \text{ 进程的工作集}) =$
最近 Δ 中所有页的引用数目 (随时间变化)
 - 如果 Δ 太小, 那么它不能包含整个局部
 - 如果 Δ 太大, 那么它可能包含多个局部
 - 如果 $\Delta = \infty$, 那么工作集合为进程执行所接触到的所有页的集合
- $D = \sum WSS_i \equiv$ 总的帧需求量
- if $D > m \Rightarrow$ 颠簸
- 策略: 如果 $D > m$, 则暂停一个进程
- 困难: 跟踪工作集

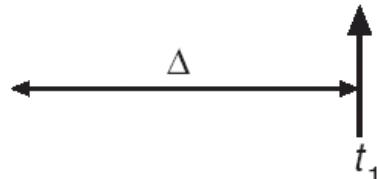




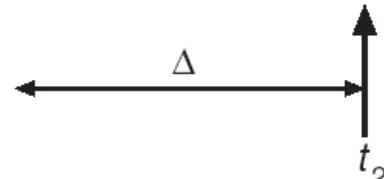
工作集模型

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 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$

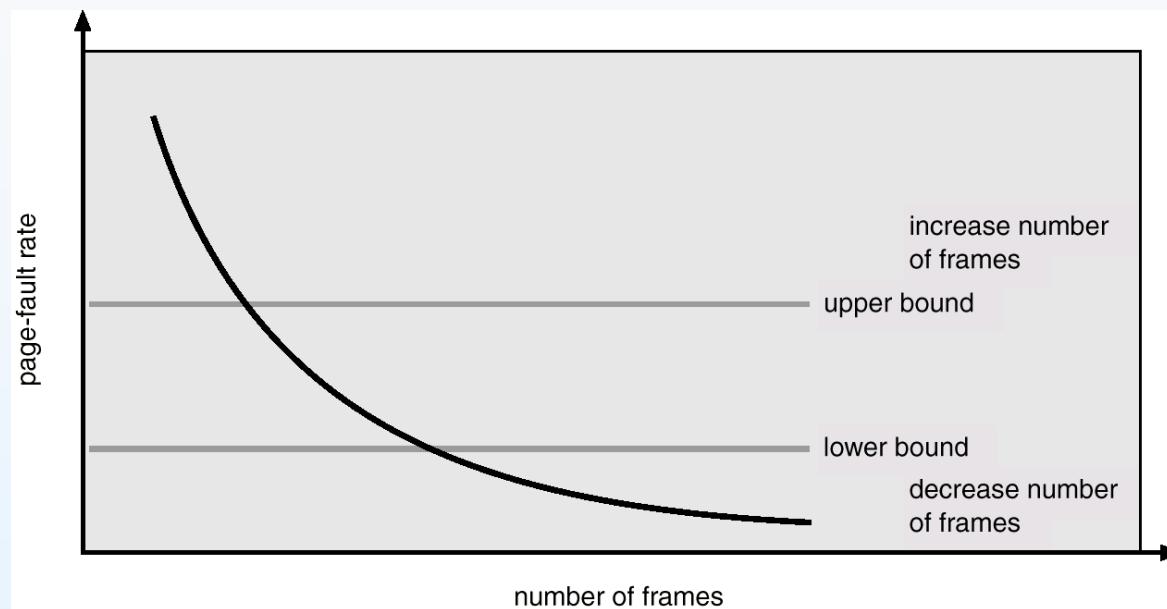


$$WS(t_2) = \{3, 4\}$$





缺页率 (PFF) 策略



■ 设置可接受的缺页率

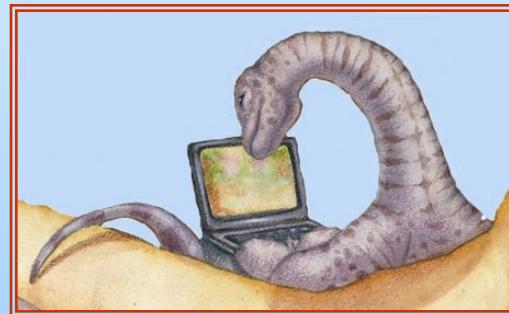
- 如果缺页率太低，回收一些进程的页框
- 如果缺页率太高，就分给进程一些页框



6、内核内存分配



伙伴系统
slab分配





内核内存分配

- 用户态进程需要内存时，可以从空闲页框链表中获得空闲页，这些页通常是分散在物理内存中的，进程最后一页可能产生内碎片

- 内核内存的分配不同于用户内存
- 通常从空闲内存池中获取，其原因是：
 1. 内核需要为不同大小的数据结构分配内存
 2. 一些内核内存需要连续的物理页





内核使用内存块的特点

■ 内核在使用内存块时有如下特点：

- (1) 内存块的尺寸比较小；
- (2) 占用内存块的时间比较短；
- (3) 要求快速完成分配和回收；
- (4) 不参与交换；
- (5) 频繁使用尺寸相同的内存块，存放同一结构的数据；
- (6) 要求动态分配和回收。





伙伴(Buddy)系统

- 主要用于Linux早期版本中内核底层内存管理
- 一种经典的内存分配方案
- 从物理上连续的大小固定的段上分配内存
- 主要思想：内存按2的幂的大小进行划分，即4KB、8KB等空闲块，组成若干空闲块链表；查找链表找到满足进程需求的最佳匹配块。三个要点：
 1. 满足要求是以2的幂为单位的
 2. 如果请求不为2的幂，则需要调整到下一个更大的2的幂
 3. 当分配需求小于现在可用内存时，当前段就分为两个更小的2的幂段，继续上述操作直到合适的段大小





Buddy 系统

■ 伙伴系统算法

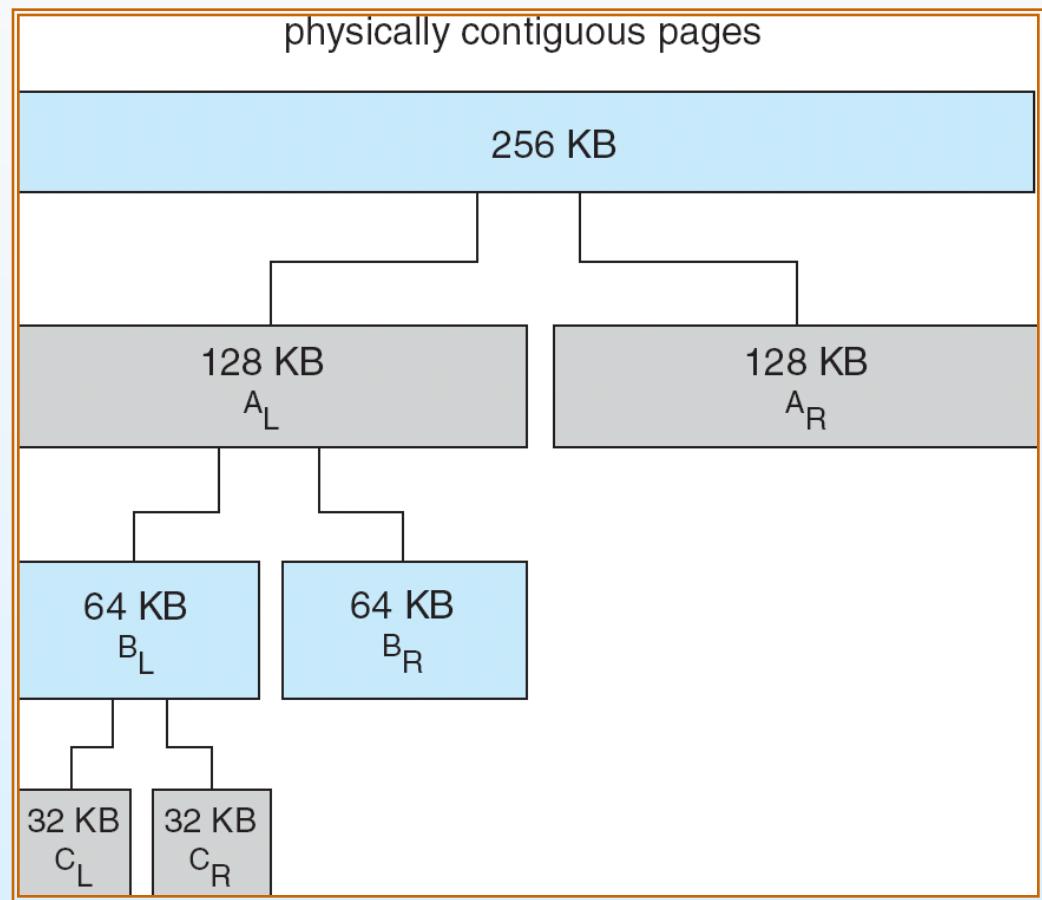
- 首先将整个可用空间看作一块，大小为 2^n
- 假设进程申请的空间大小为 s ，如果满足 $2^{n-1} < s \leq 2^n$ ，则分配整个块，否则将块划分为两个大小相等的伙伴，大小为 2^{n-1}
- 一直划分下去直到产生大于或等于 s 的最小块分配给进程

■ 优点

- 可通过合并而快速形成更大的段

■ 缺点

- 调整到下一个 2 的幂容易产生碎片





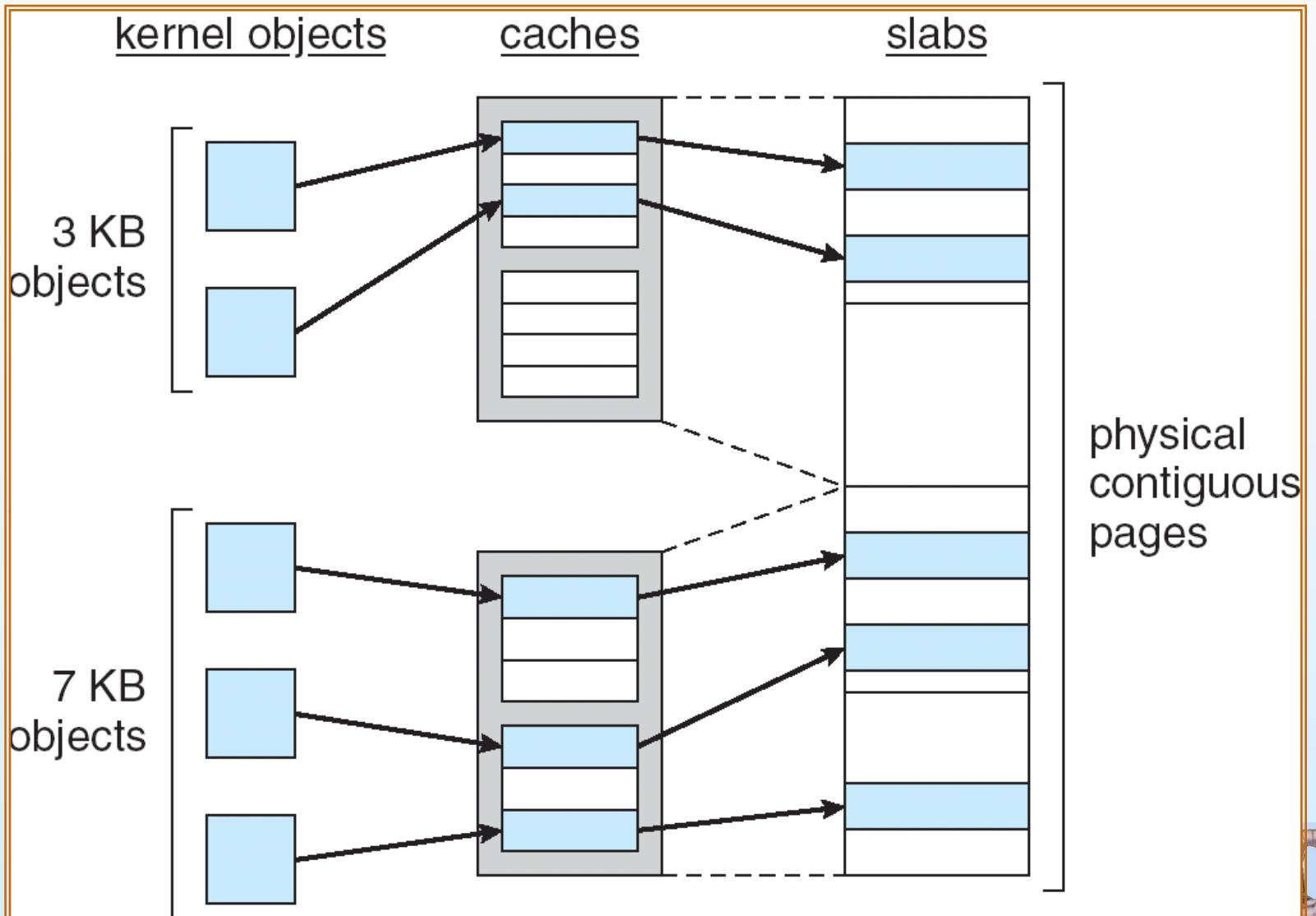
Slab 分配

- 内核分配的另一方案
- **Slab** 也称为板块，是由一个或多个物理上连续的页或内存卡组成，**Slab**中的内存块需要一起建立或撤销
- 高速缓存**Cache**，又称为板块组，含有一个或多个slab；系统具有多个cache，分别对应多种尺寸和结构相同的内存块
- 每个cache都有内核数据结构的对象，如进程描述符、文件对象、信号量等。
 - 每个 cache 含有内核数据结构的对象实例。例如信号量 cache存储着信号量对象，进程描述符cache存储着进程描述符对象。





Slab Allocation





Slab 分配

- 当创建 **cache** 时，包括若干个标记为空闲的对象，对象的数量与 **slab** 的大小有关
 - 12KB的**slab**（包括3个连续的页）可以存储6个2KB大小的对象。开始所有的对象都标记为空闲。
 - 当需要内核对象时，可从**cache**上直接获取，并标识对象为已使用。
 - 例如：在Linux系统中，进程描述PCB的类型为**struct task_struct**，大小为1.7KB。当Linux内核创建新任务时，它会从**cache**中获得**task_struct**对象所需要的内存。**Cache**上会有已分配好的并标记为空闲**task_struct**对象来满足。
- Slab有三种状态：
 - 满的： **slab**中所有对象被标记为使用
 - 空的： **slab**中所有对象被标记为空闲
 - 部分： **slab**中有的对象被标记为使用，有的对象被标记为空闲
- 当一个**slab**充满了已使用的对象时，下一个对象的分配从空的**slab**开始分配
 - 如果没有空的**slab**，则从物理连续页上分配新的**slab**，并赋给一个**cache**





Slab 分配

■ 优点

- ① 没有因碎片而引起的内存浪费

因为每个内核数据结构都有相应的cache，而每个cache都由若干slab组成，每个slab又分为若干与对象大小相同的部分

- ② 内存请求可以快速满足

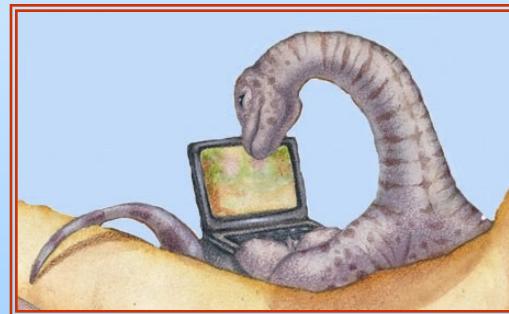
由于对象预先创建，所以可以快速分配，刚用完对象并释放时，只需要标记为空闲并返回，以便下次使用

■ 使用的系统

- Solaris 2.4内核 最先应用
- Solaris某些用户态内存请求
- Linux 2.2 以后的内核



7、其它考慮





内容

- 预调页
- 页大小
- TLB范围
- 反向页表
- I/O互锁
- 程序结构



预先调页

- 在进程启动初期，减少大量的缺页中断
 - 例如：当重启一个换出进程时，由于所有的页都在磁盘上，每个页都必须通过缺页中断调入内存
- 在引用前，调入进程的所有或一些需要的页面
 - 例如：对于采用工作集的系统，为每个进程保留一个位于其工作集内的页的列表。
- 如果预调入的页面没有被使用，则内存被浪费





页面尺寸选择

- 页面大小总是2的幂，通常是**4KB-4MB**
- 页表大小-需要大的页
 - 对于给定的虚拟内存空间，降低页大小，也就是增加了页的数量，因此也增加了页表大小
 - 例如：**4MB**的虚拟内存，如果页大小为**1KB**，那么就有**4096**页，如果页大小为**8KB**，那么只有**512**页
- 碎片 - 需要小的页
 - 较小的页可能更好的利用内存，业内碎片
- I/O 开销 - 需要大的页
 - I/O时间包括寻到、延迟和传输时间，尽管传输时间和传输量（即页的大小）成正比，需要小的页，但是寻道时间和延迟时间远远超过传输时间





页面尺寸选择

■ 程序局部 – 需要小的页

- 较小的页允许每个页更精确匹配程序局部，而采用较大的页不但传输所需要的，还会传输在页内的其他不需要使用的内容

■ 缺页次数 – 需要大的页

- 由于每个缺页会产生大量的额外开销，为了降低缺页次数，需要较大的页

■ 其他因素

- 如页大小和调页设备的扇区大小的关系等

■ 没有最佳答案，总的来说，趋向更大的页

■ 1983年，4KB为页大小的上限

■ 1990年，4KB最常用的页大小

■ 现在，Linux，4MB





TLB 范围

- TLB 范围 – 通过**TLB**所访问的内存量
- $\text{TLB \ 范围} = (\text{TLB \ 大小}) \times (\text{页大小})$
- 理想情况下，一个进程的工作集应存放在 **TLB**中，否则会有大量的缺页中断
 - 如果把**TLB**条数加倍，那么**TLB**的范围就加倍，但是对于某些使用大量内存的应用程序，这样做可能不足以存放工作集
- 增加页的大小
 - 对于不需要大页的应用程序而言，这将导致碎片的增加
- 提供多种页的大小
 - 例如UltraSPARC芯片支持8KB、64KB、512KB、4MB大小的页
 - Solaris使用了8KB和4MB大小的页，对于具有64项的TLB，Solaris的TLB范围可从512KB（使用8KB的页）到256M（使用4MB的页）
 - 对于大部分程序，8KB的大小足够了，对于需要大页的应用程序（如数据库）有机会使用大页而不增加碎片的大小





反向页表

- 反向页表降低了保存的物理内存
- 不再包括进程逻辑地址空间的完整信息
- 为了提供这种信息，进程必须保留一个外部页表
- 外部页表可根据需要换进或换出内存





程序结构

■ 数据结构和程序结构可能影响系统性能

- array A[1024, 1024] of integer
- 每行保存在一页
- 分配一个页框

- 程序1: 1024×1024 缺页
- 程序2: 1024 次缺页

■ 其它因素（编译器、载入器、程序设计语言）对调页都有影响

程序1:

```
for j := 1 to 1024 do  
  for i := 1 to 1024 do  
    A[i,j] := 0;
```

程序2:

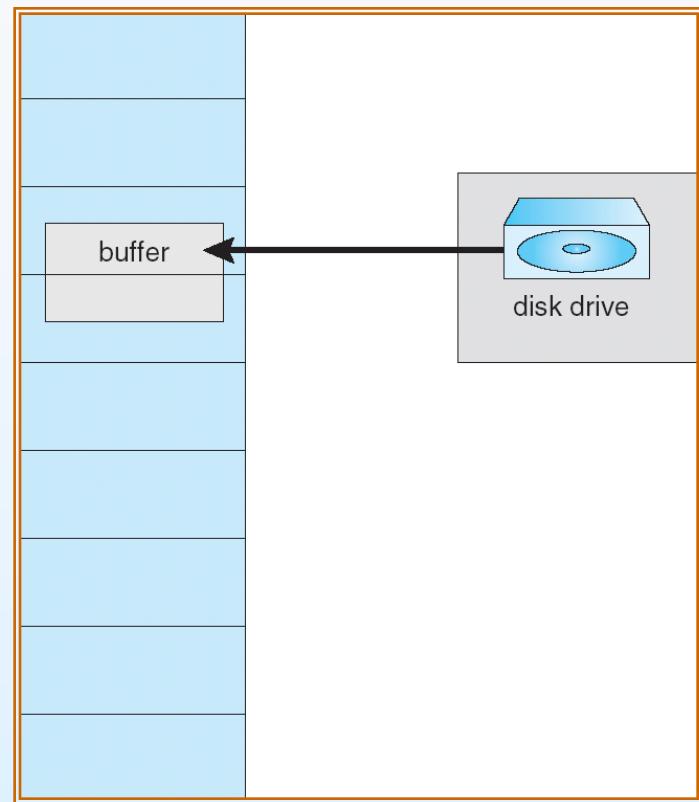
```
for i := 1 to 1024 do  
  for j := 1 to 1024 do  
    A[i,j] := 0;
```





I/O 互锁

- 允许某些页在内存中被锁住
- 为了防止I/O出错，有两种解决方案：
 - 不对用户内存进行I/O，即I/O只在系统内存和I/O设备间进行，数据在系统内存和用户内存间复制
 - 允许页所在内存，锁住的页不能被置换，即正在进行I/O的页面不允许被置换算法置换出内存，当I/O完成时，页被解锁



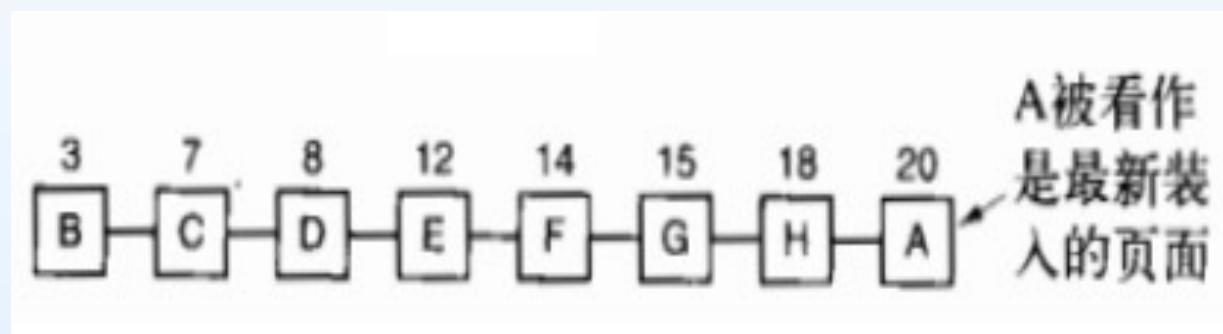


1. 虚拟存储管理系统的基础是程序的（）理论。
1. 在请求页式存储管理中，产生缺页中断是因为查找的页不在（）中。
2. 虚存管理和实存管理的主要区别是（）。





■ 考虑下图中的页面序列。假设从页面B到页面A的R位分别是11011011。使用第二次机会算法，被移走的是页面（）。





在分页存储管理中，减少页面大小，可以减少内存的浪费。所以页面越小越好。

当采用分页式虚拟存储管理时，如果在进程执行过程中需访问的页面为无效时，硬件将发出一个缺页中断。





- 1、实现进程间数据共享最方便的存储管理技术是什么？
- 2、页表和空闲页表的区别是什么？
- 3、程序的页面大小和内存的帧大小不一致，会有什么问题？

