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 $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_i = 10$

$s_2 = 127$

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

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

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

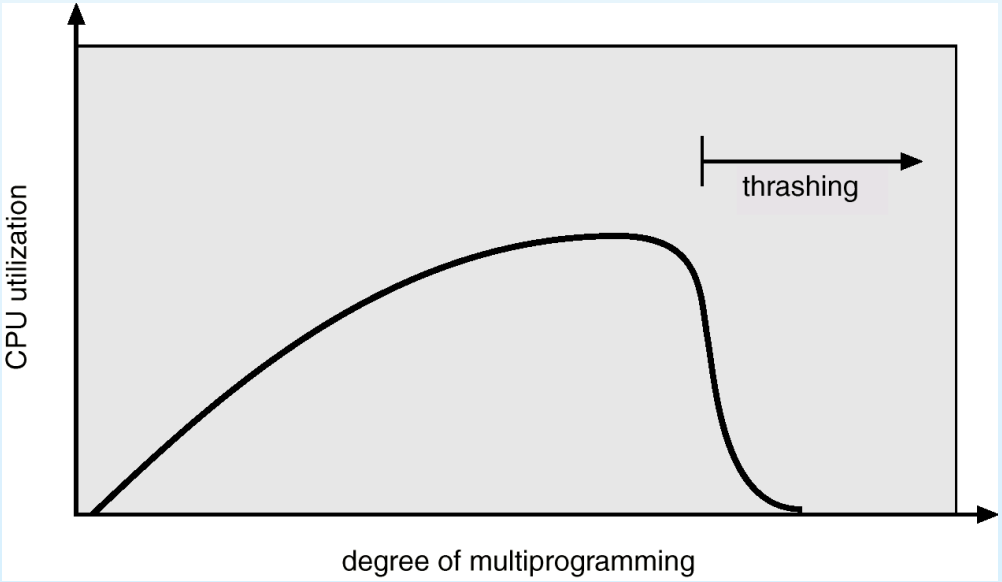
全局置换和局部置换

- 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 ≡ 一个进程的页面经常换入换出



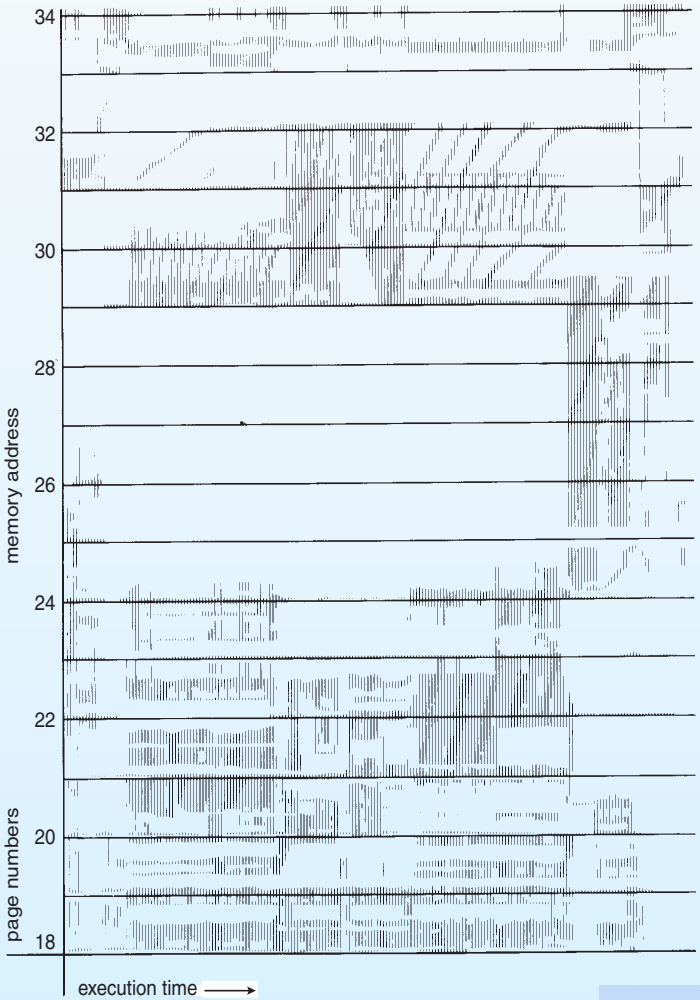
引发这种现象的原因：系统内存不足，页面置换算法不合理

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

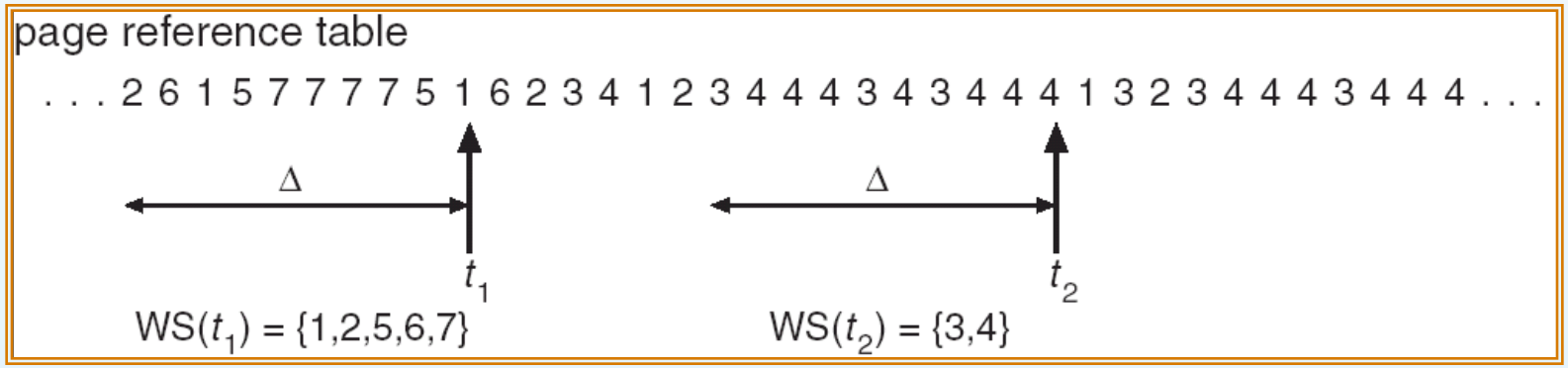
- 为什么颠簸会发生
  Σ size of locality > total memory size



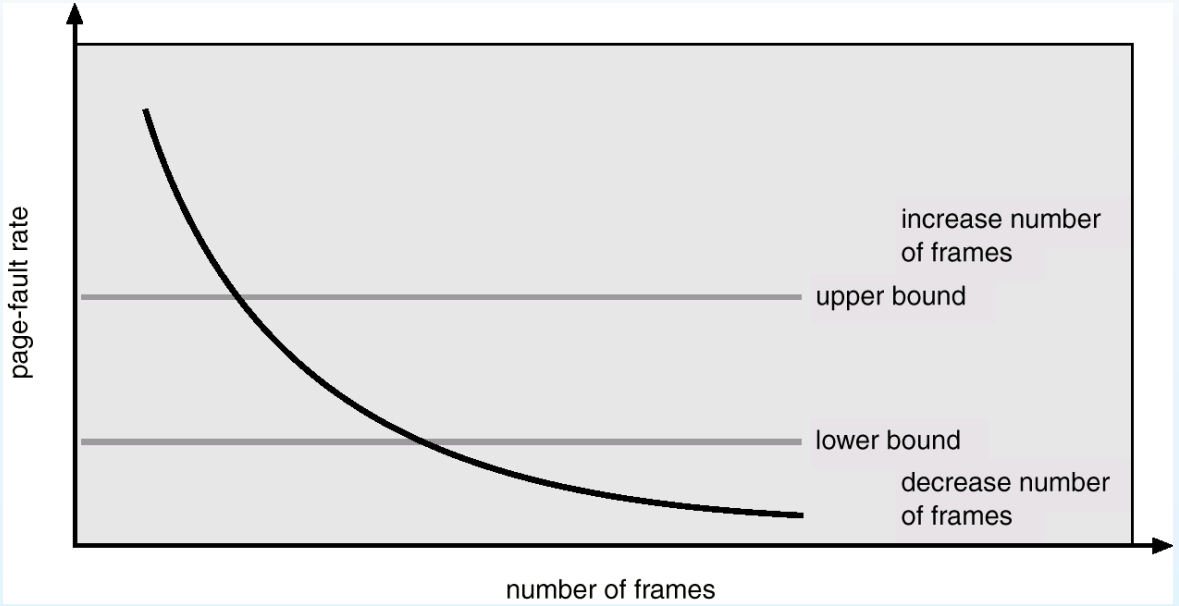
工作集模型(working-set model)

- Δ ≡ 工作集窗口 ≡ 固定数目的页的引用
  例如: 10,000 个引用
- WSS_i (P_i 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
- $D = \Sigma\ WSS_i$ ≡ 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 . . .
Δ
Δ
t_1
t_2
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；

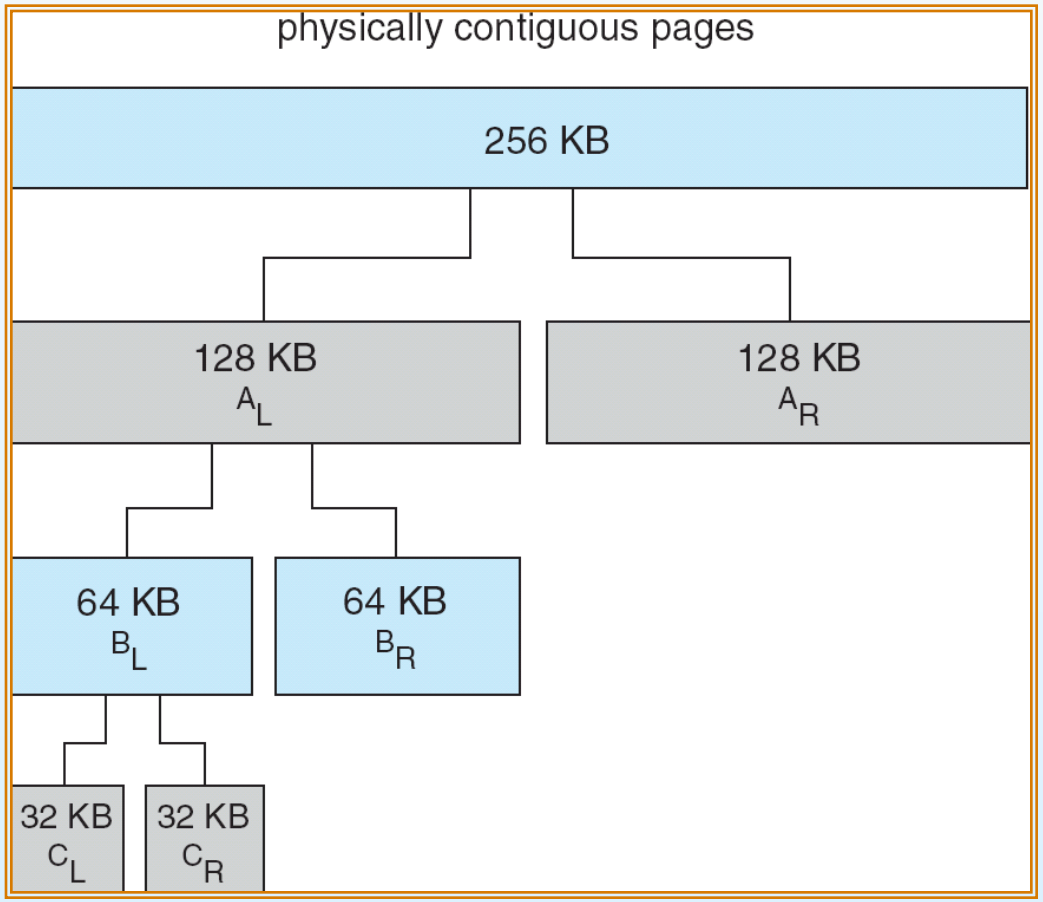
  (6)要求动态分配和回收。

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

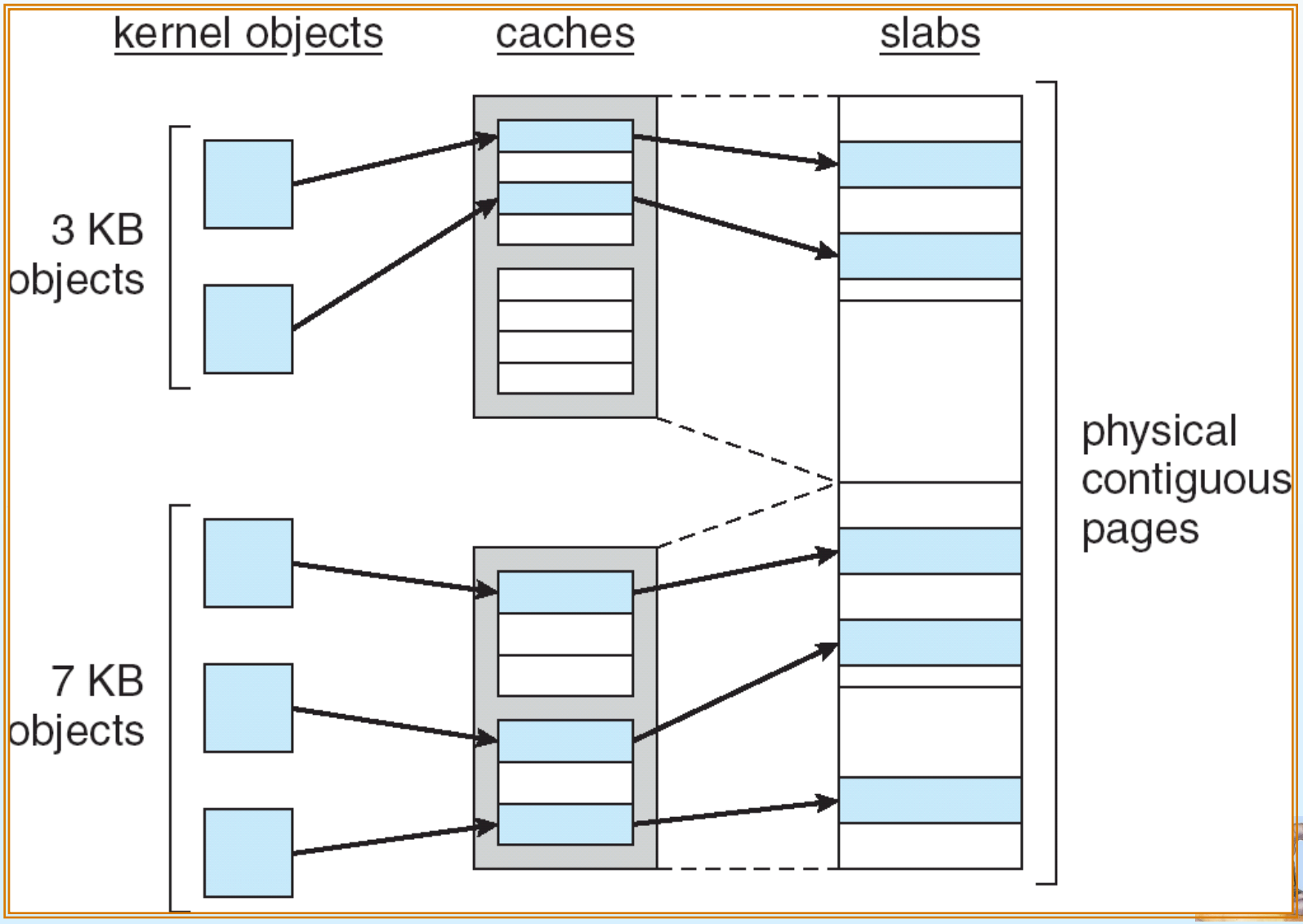
- 伙伴系统算法
  - 首先将整个可用空间看作一块，大小为2^n
  - 假设进程申请的空间大小为s，如果满足$2^{n-1}<s<2^n$，则分配整个块，否则将块划分为两个大小相等的伙伴，大小为2^{n-1}
  - 一直划分下去直到产生大于或等于s的最小块分配给进程
- 优点
  - 可通过合并而快速形成更大的段
- 缺点
  - 调整到下一个2的幂容易产生碎片



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
- 每个内核数据结构都有一个cache，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，小大为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、其它考虑

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
- TLB 范围 = (TLB 大小) X (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 x 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