操作系统　Operating System

第三章 内存管理(6)

沃天宇
woty@buaa.edu.cn
2021年3月30日

页面置换策略（Replacement Strategies）

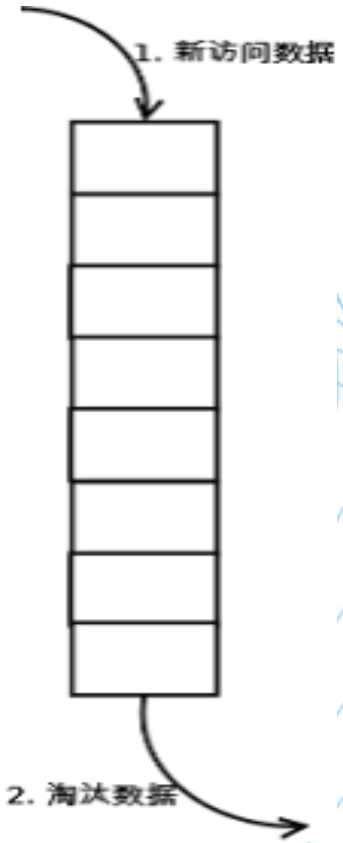
- 1. 一种最佳策略：从主存中移出永远不再需要的页面，如无这样的页面存在，则应选择最长时间不需要访问的页面。
- 2. 先进先出算法（First-in, First-out）：总选择作业中在主存驻留时间最长的一页淘汰。
- 3. 最近最久不用的页面置换算法（Least Recently Used Replacement）：当需要置换一页面时，选择在最近一段时间内最久不用的页面予以淘汰。

最优置换（optimal page-replacement）

- 是所有页置换算法中页错误率最低的，但它需要引用先验知识，因此无法被实现。
- 它会将内存中的页 P 置换掉，页 P 满足：从现在开始到未来某刻再次需要页 P，这段时间最长。也就是 OPT 算法会置换掉未来最久不被使用的页 。
- OPT 算法通常用于比较研究，衡量其他页置换算法的效果。

先进先出（First-in, First-out）

- 最简单的页置换算法，操作系统记录每个页被调入内存的时间，当必需置换掉某页时，选择最旧的页换出。具体实现如下：
  - 新访问的页面插入FIFO队列尾部，页面在FIFO队列中顺序移动；
  - 淘汰FIFO队列头部的页面；
- 性能较差。较早调入的页往往是经常被访问的页，这些页在FIFO算法下被反复调入和调出。并且有Belady现象。

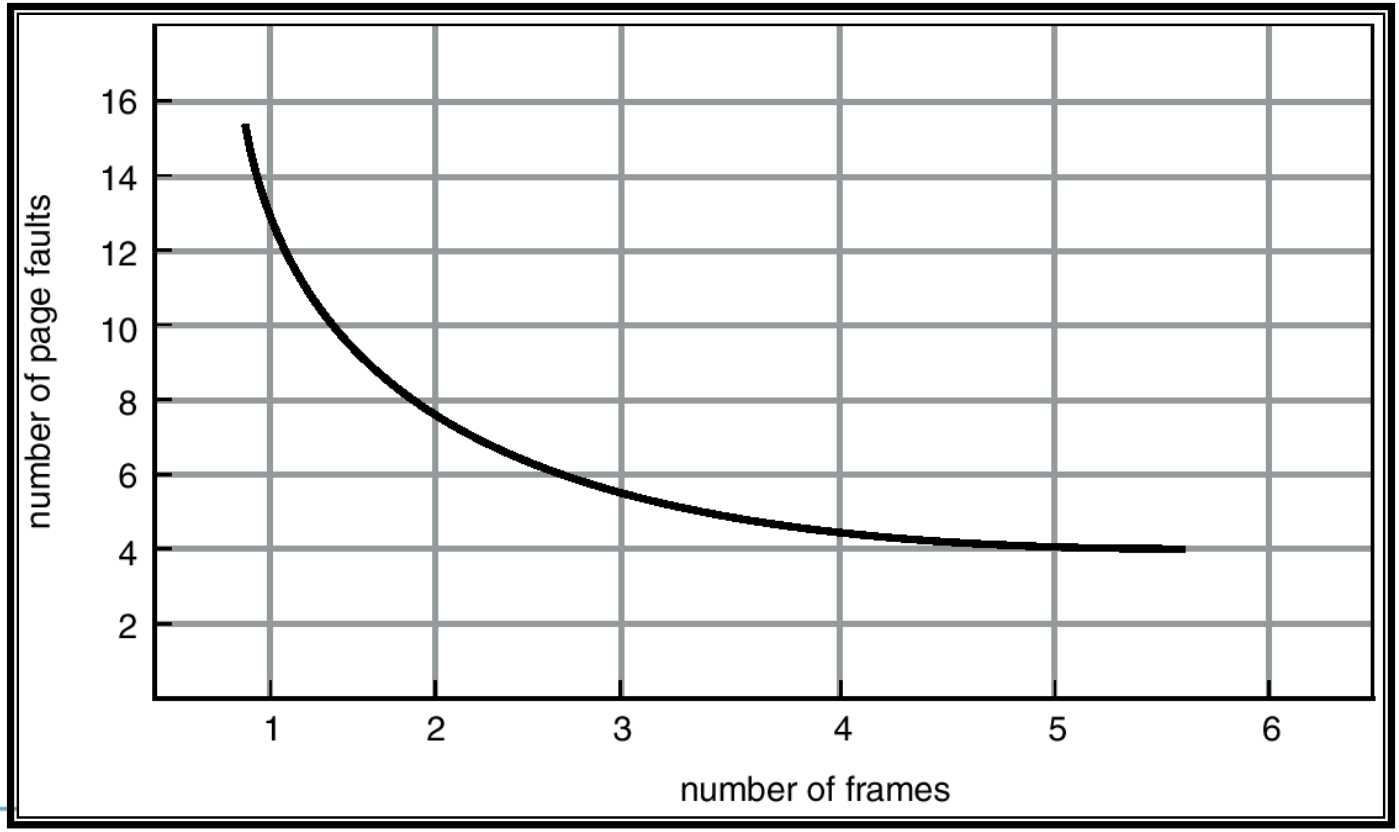


Belady现象

- 在使用FIFO算法作为缺页置换算法时，分配的页面增多，但缺页率反而提高，这样的异常现象称为Belady Anomaly。
- 虽然这种现象说明的场景是缺页置换，但在运用FIFO算法作为缓存算法时，同样也是会遇到，增加缓存容量，但缓存命中率也会下降的情况。

理想的情况

- 理想的情况：缺页率随页框数增加而下降



Belady现象

- 进程P有5页程序访问页的顺序为：
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5；
- 如果在内存中分配3个页面，则缺页情况如下：12次访问中有缺页9次；

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	1	2	5	5	5	3	4	4
页 1		1	2	3	4	1	2	2	2	5	3	3
页 2			1	2	3	4	1	1	1	2	5	5
缺 页	x	x	x	x	x	x	x	√	√	x	X	√

Belady现象

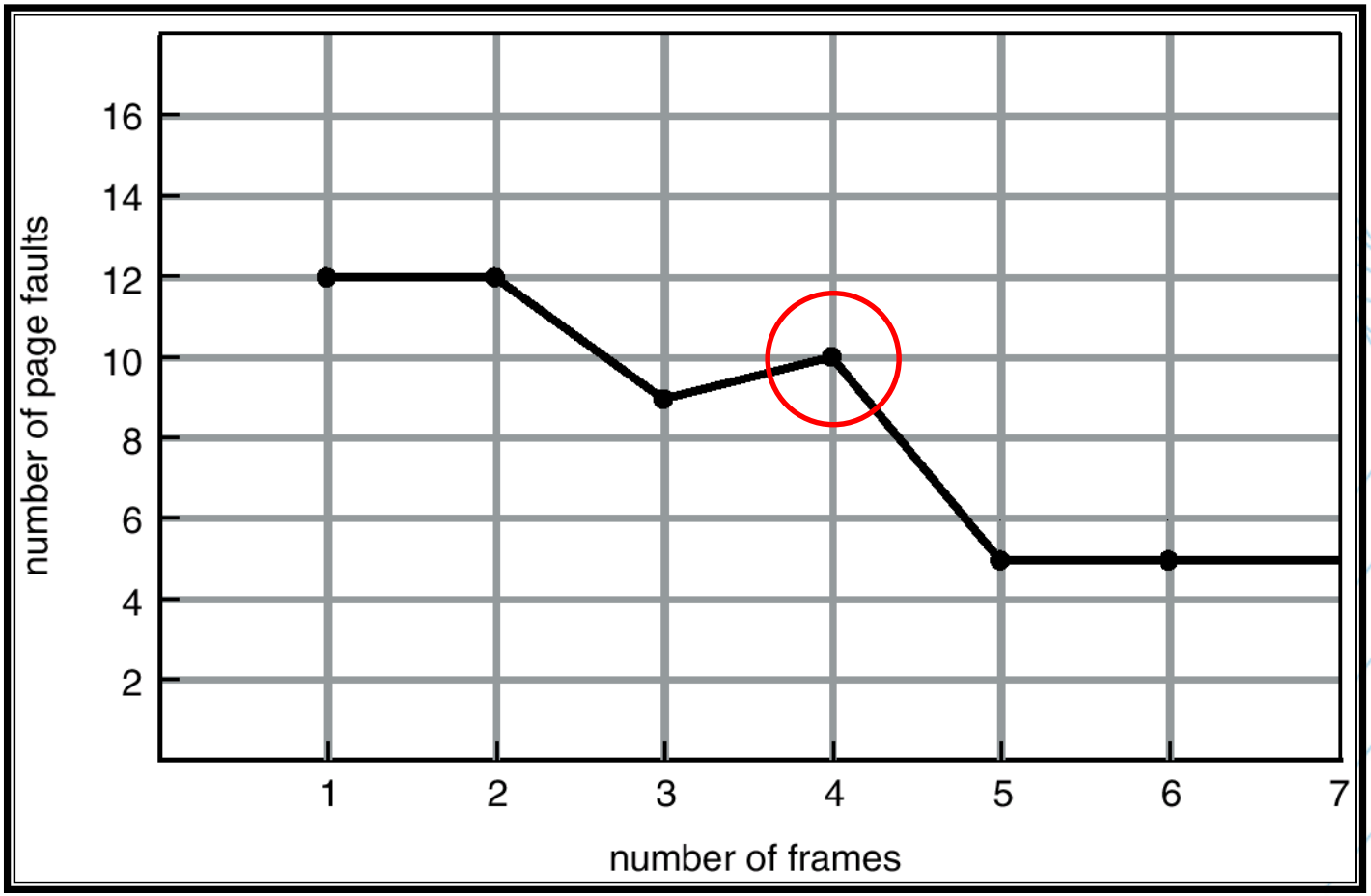
- 如果在内存中分配4个页面，则缺页情况如下：12次访问中有缺页10次；

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	4	4	5	1	2	3	4	5
页 1		1	2	3	3	3	4	5	1	2	3	4
页 2			1	2	2	2	3	4	5	1	2	3
页 3				1	1	1	2	3	4	5	1	2
缺 页	x	x	x	x	√	√	x	x	x	x	x	x

Belady现象：分配的页面数增多但缺页率反而提高的异常现象

反应出： FIFO算法的置换特征与进程访问内存的动态特征是矛盾的

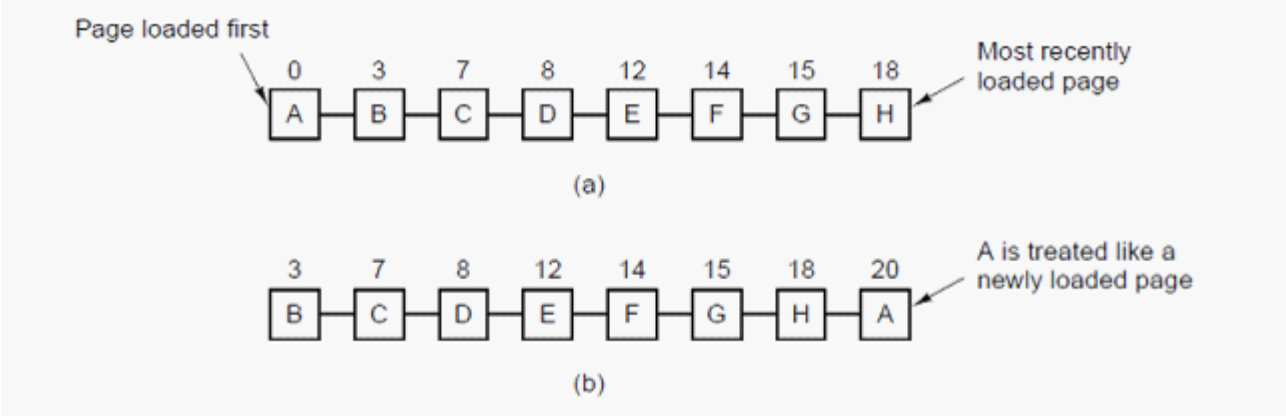
FIFO Illustrating Belady's Anamoly



改进的FIFO算法—Second Chance

其思想是“如果被淘汰的数据之前被访问过，则给其第二次机会（Second Chance）”实现：

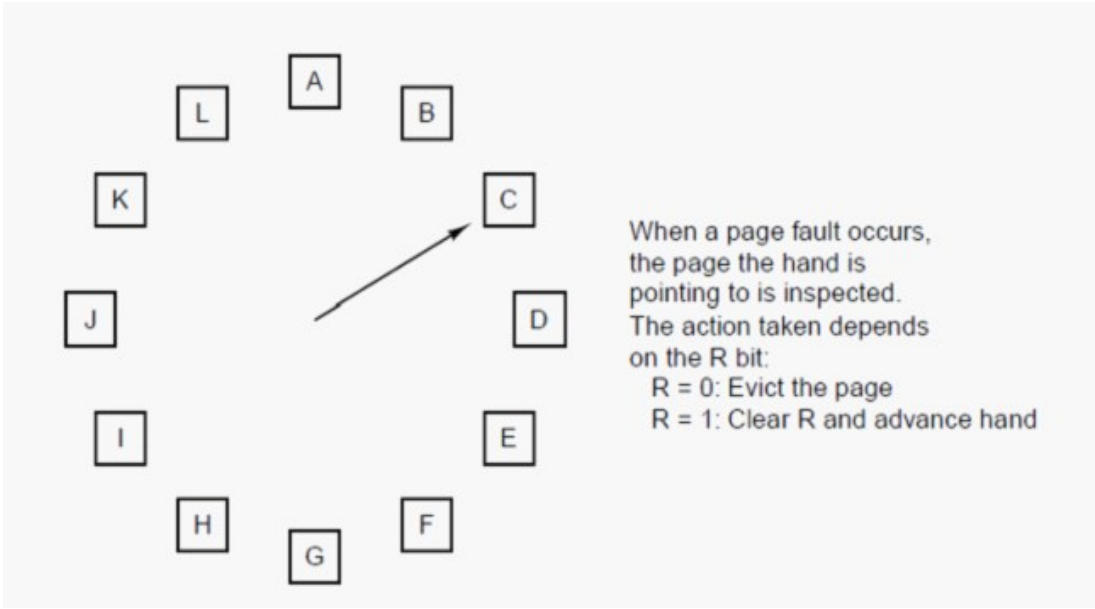
- 每个页面会增加一个访问标志位，用于标识此数据放入缓存队列后是否被再次访问过。



- A是FIFO队列中最旧的页面，且其放入队列后没有被再次访问，则A被立刻淘汰；否则如果放入队列后被访问过，则将A移到FIFO队列头，并且将访问标志位清除。
- 如果所有的页面都被访问过，则经过一次循环后就会按照FIFO的原则淘汰。

改进的FIFO算法— Clock

- Clock是Second Chance的改进版，也称最近未使用算法（NRU, Not Recently Used）。通过一个环形队列，避免将数据在FIFO队列中移动。算法如下：
  - 产生缺页错误时，当前指针指向C，如果C被访问过，则清除C的访问标志，并将指针指向D；
  - 如果C没有被访问过，则将新页面放入到C的位置，置访问标志，并将指针指向D。



FIFO类算法对比

对比点	对比
命中率	Clock = Second Chance > FIFO
复杂度	Second Chance > Clock > FIFO
代价	Second Chance > Clock > FIFO

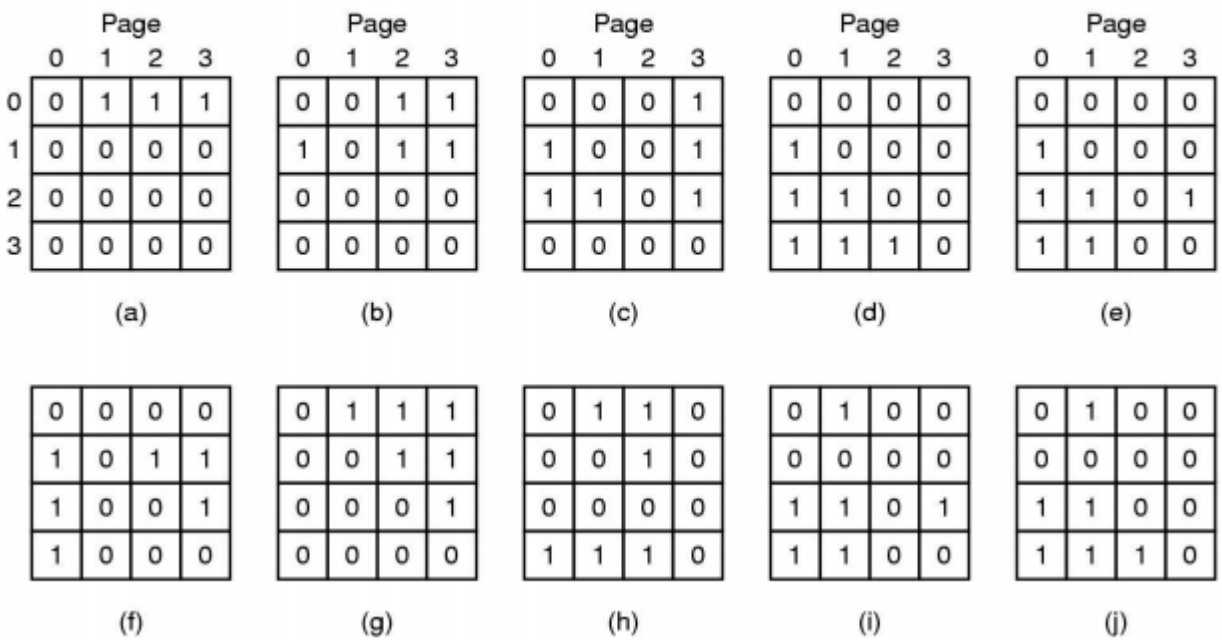
由于FIFO类算法命中率相比其他算法要低不少，因此实际应用中很少使用此类算法。

最近最少使用（Least recently used）

- LRU算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。
- 这是局部性原理的合理近似，性能接近最优算法。但由于需要记录页面使用时间的先后关系，硬件开销太大。方法之一：
  - 设置一个特殊的栈，保存当前使用的各个页面的页面号。
  - 每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。栈底始终是最近最久未使用页面的页面号。

LRU的一个硬件实现

- 页面访问顺序0, 1, 2, 3, 2, 1, 0, 3, 2, 3

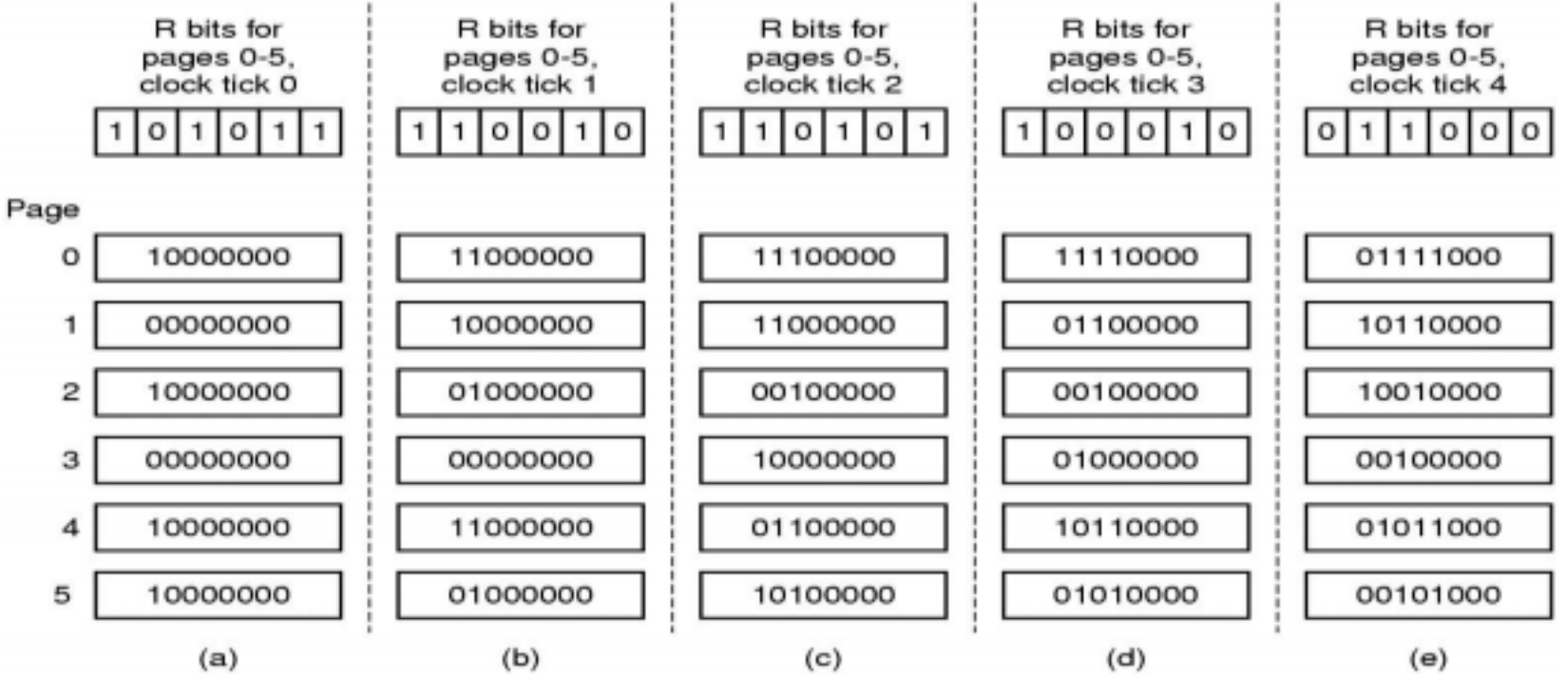


- **说明：** 访问第i页时先将页所在行置1，所在列清0。将行编号最小的那一页置换出去。本例中(j)处若发生页置换，则应置换第1页。

老化算法（AGING）

LRU算法开销很大，硬件很难实现。老化算法是LRU的简化，但性能接近LRU:

- 为每个页面设置一个移位寄存器，并设置一位访问位R，每隔一段时间，所有寄存器右移1位，并



算法举例(Opt, LRU, FIFO, Clock)

- 3个页框，5个工作页面

其它替换算法

页面替换算法还有很多：

- LRU类：LRU2， Two queues（2Q）， Multi Queue（MQ）；
- LFU类： LFU（Least Frequently Used）， LFU-Aging， LFU*-Aging， Window-LFU；
- 其它： Most Recently Used（MRU），Adaptive Replacement Cache（ARC），Working Set（WS），Working Set Clock（WSclock）

感兴趣的同学可以在网上查找相关算法并自学

页面置换算法对比

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

《现代操作系统》P121

http://en.wikipedia.org/wiki/Page_replacement_algorithm

更新问题

在虚存系统中，主存作为辅存（磁盘）的高速缓存，保存了磁盘信息的副本。因此，当一个页面被换出时，为了保持主辅存信息的一致性，必要时需要信息更新：

- 若换出页面是file backed类型，且未被修改，则直接丢弃，因为磁盘上保存有相同的副本。
- 若换出页面是file backed的类型，但已被修改，则直接写回原有位置。
- 若换出页面是anonymous类型，若是第一次换出且未被修改，则写入Swap区，若非第一次则丢弃。
- 若换出页面是anonymous类型，且已被修改，则写入Swap区。

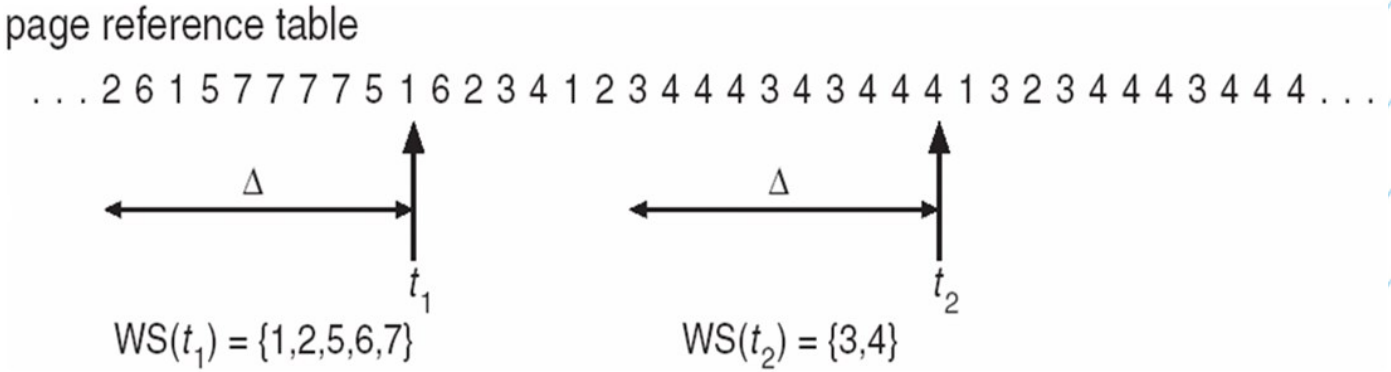
工作集与驻留集管理

前面我们从一个进程的角度，讨论了虚存管理中的相关问题，下面我们将从系统管理者的角度（OS的视角）讨论多个进程同时存在，虚存管理中的问题。

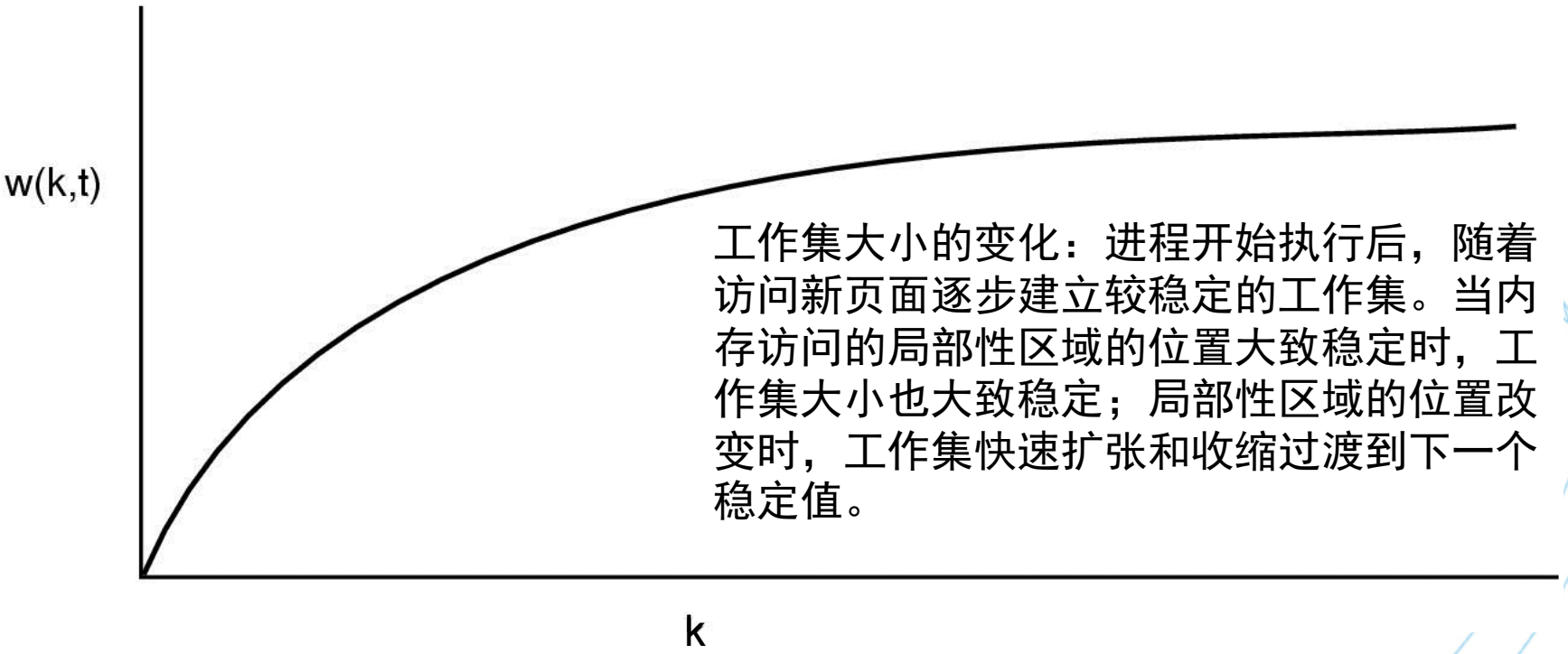
- 进程的工作集（working set）：当前正在使用的页面的集合。
- 进程的驻留集（Resident Set ）：虚拟存储系统中，每个进程驻留在内存的页面集合，或进程分到的物理页框集合。

工作集策略(working set strategy)

- 引入工作集的目的是依据进程在过去的一段时间内访问的页面来调整常驻集大小。
  - 定义：工作集是一个进程执行过程中所访问页面的集合，可用一个二元函数WS(k,t)表示，其中：t是执行时刻；k是窗口尺寸(window size)；工作集是在[t - k, t]时间段内所访问的页面的集合，
    w(k,t)=|WS(k,t)| 指工作集大小即页面数目；



Behavior of working set as a function of k



W(k,t) is number of pages at time t used by k most recent memory references

获得精确工作集的困难

- 工作集的过去变化未必能够预示工作集的将来大小或组成页面的变化；
- 记录工作集变化要求开销太大；
- 对工作集窗口大小Δ的取值难以优化，而且通常该值是不断变化的；

驻留集的管理

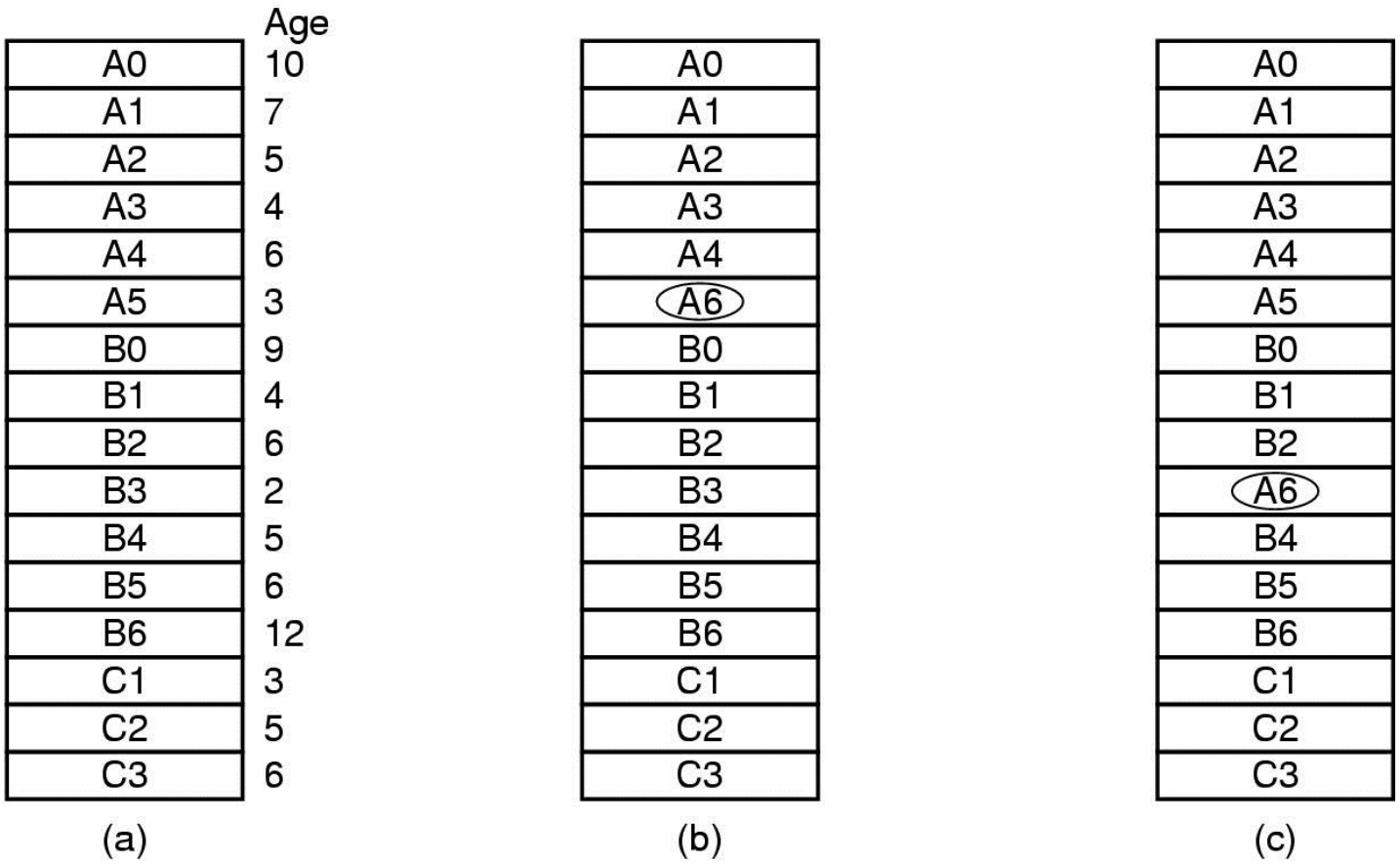
- 进程驻留集管理主要解决的问题是，系统应当为每个活跃进程分配多少个页框。
- 影响页框分配的主要因素：分配给每个活跃进程的页框数越少，同时驻留内存的活跃进程数就越多，进程调度程序能调度就绪进程的概率就越大。然而，这将导致进程发生缺页中断的概率较大；为进程分配过多的页框，并不能显著地降低其缺页中断率。

分配和置换策略

- 考虑进程划分的额外约束
  - 固定分配局部置换
  - 可变分配全局置换
  - 可变分配局部置换

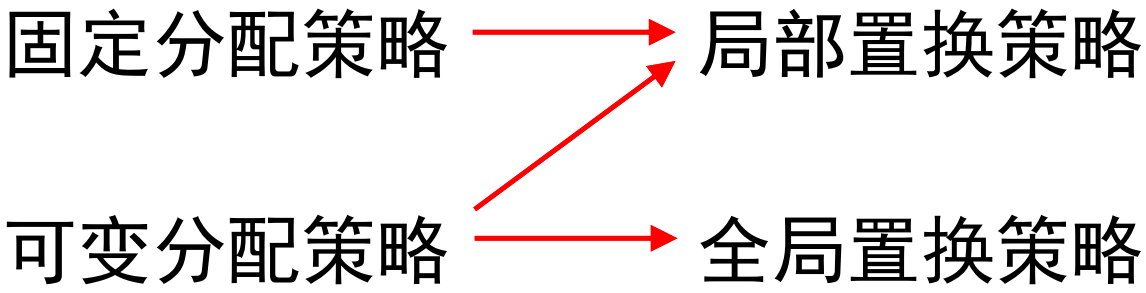
教材P188

分配和置换策略



初始页面配置　　局部页面置换　　全局页面置换

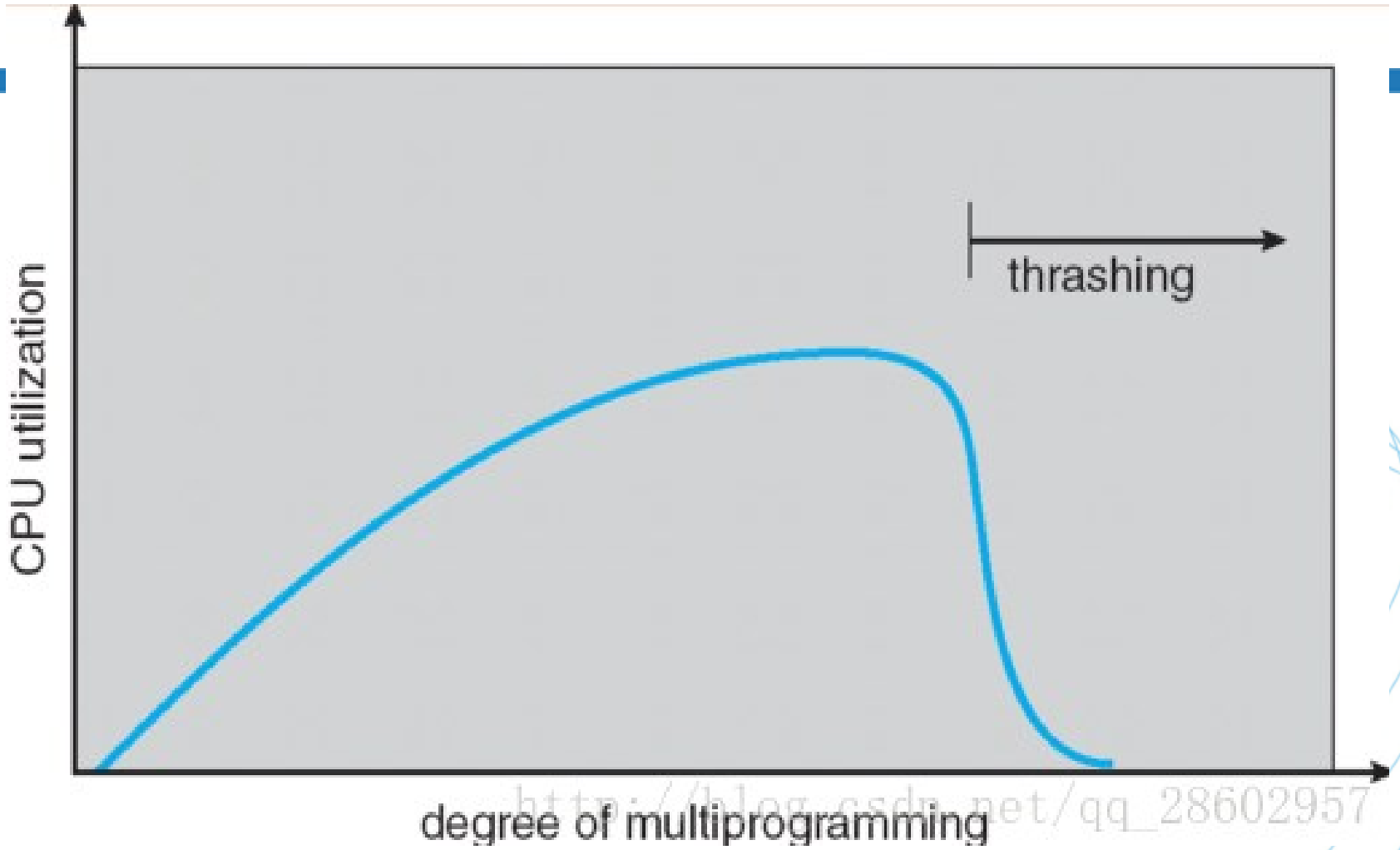
分配模式与置换模式的搭配



- 全局置换算法存在的一个问题是，程序无法控制自己的缺页率。一个进程在内存中的一组页面不仅取决于该进程的页面走向，而且也取决于其他进程的页面走向。因此，相同程序由于外界环境不同会造成执行上的很大差别。使用局部置换算法就不会出现这种情况，一个进程在内存中的页面仅受本进程页面走向的影响。
- 可变分配策略+局部置换策略：可增加或减少分配给每个活跃进程的页框数；当进程的页框全部用完，而需要装入一个新的页面时，系统将在该进程的当前驻留集中选择一个页面换出内存。

抖动问题(thrashing)

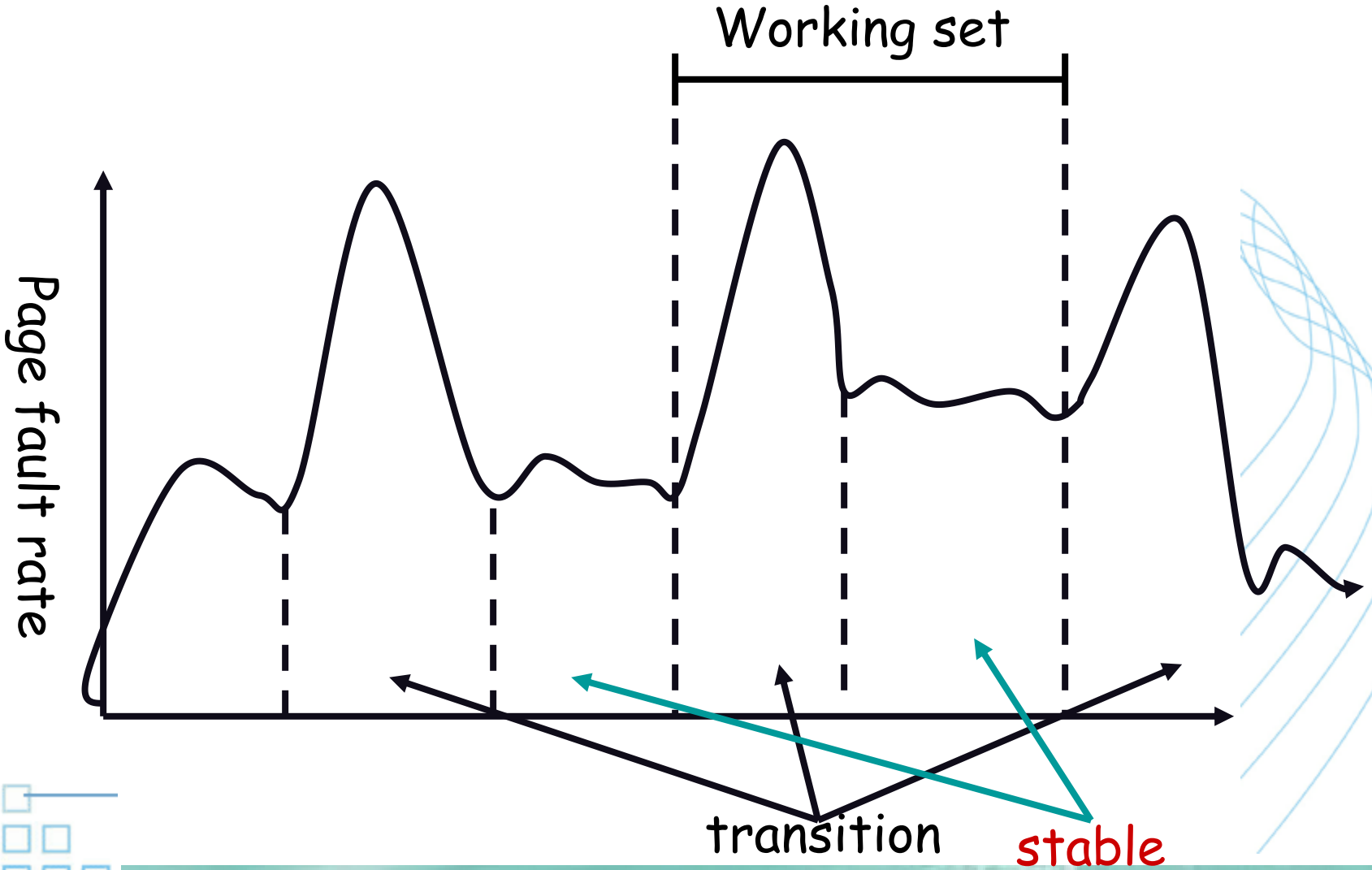
- 随着驻留内存的进程数目增加，或者说进程并发水平(multiprogramming level)的上升，处理器利用率先是上升，然后下降。
- 这里处理器利用率下降的原因通常称为虚拟存储器发生“抖动”，也就是：每个进程的常驻集不断减小，缺页率不断上升，频繁调页使得调页开销增大。
- OS要选择一个适当的进程数目，以在并发水平和缺页率之间达到一个平衡。



抖动的预防与消除

- 局部置换策略
- 引入工作集算法
- 预留部分页面
- 挂起若干进程

工作集与缺页率



改善时间性能的途径

- 降低缺页率
  - 缺页率越低，虚拟存储器的平均访问时间延长得越小；
- 提高外存的访问速度
  - 外存和内存的访问时间比值越大，则达到同样的时间延长比例，所要求的缺页率就越低；
- 高速缓存命中率。

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

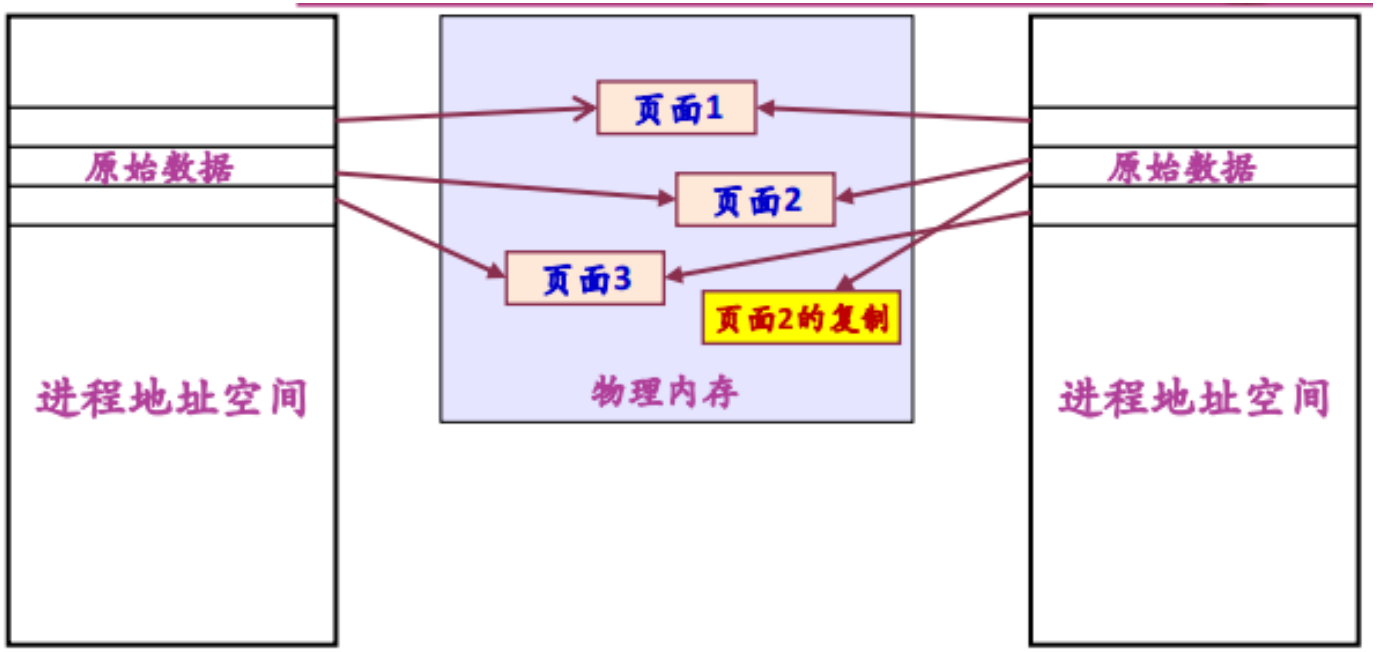
- Effective Access Time (EAT)

$$\begin{aligned} EAT = {} & (1 - p) \times \text{memory access} \\ & + p \text{ (page fault overhead} \\ & + [\text{swap page out }] \\ & + \text{swap page in} \\ & + \text{restart overhead)} \end{aligned}$$

虚拟内存其他用途

- 写时拷贝（Copy-on-Write）
- 内存映射文件（Memory-Mapped Files）

写时复制技术



- 两个进程共享同一块物理内存，每个页面都被标志成了写时复制。共享的物理内存中每个页面都是只读的。如果某个进程想改变某个页面时，就会与只读标记冲突，而系统在检测出页面是写时复制的，则会在内存中复制一个页面，然后进行写操作。新复制的页面对执行写操作的进程是私有的，对其他共享写时复制页面的进程是不可见的。

写时复制技术(copy-on-write)

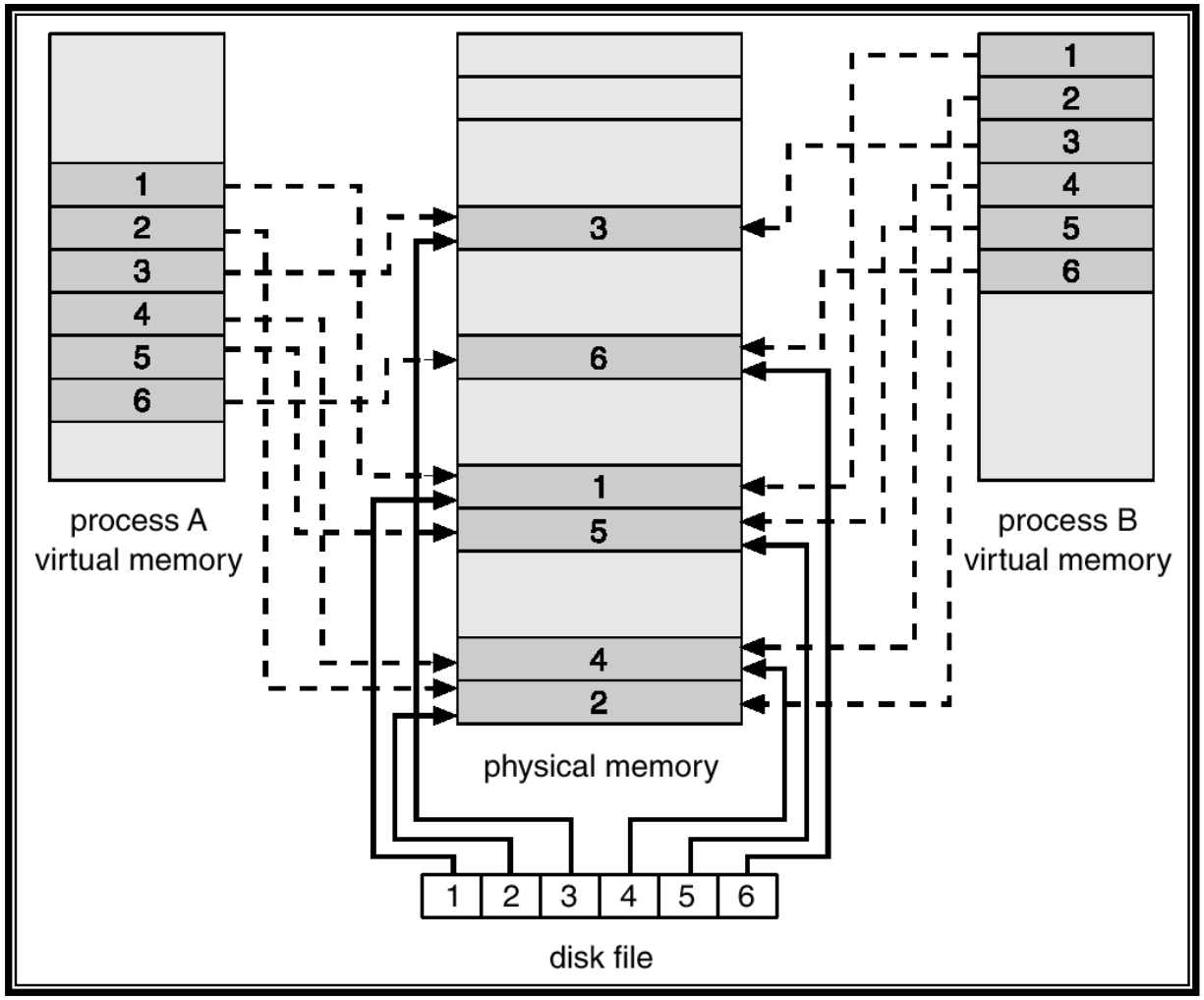
写时复制的优点

- 传统的fork()系统调用直接把所有的资源复制给新创建的进程。这种实现过于简单而效率低，因为它拷贝的数据也许并不共享。
- Linux的fork()使用写时拷贝(copy-on-write)实现，它可以推迟甚至免除拷贝数据的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会复制，从而使各个进程都拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行。

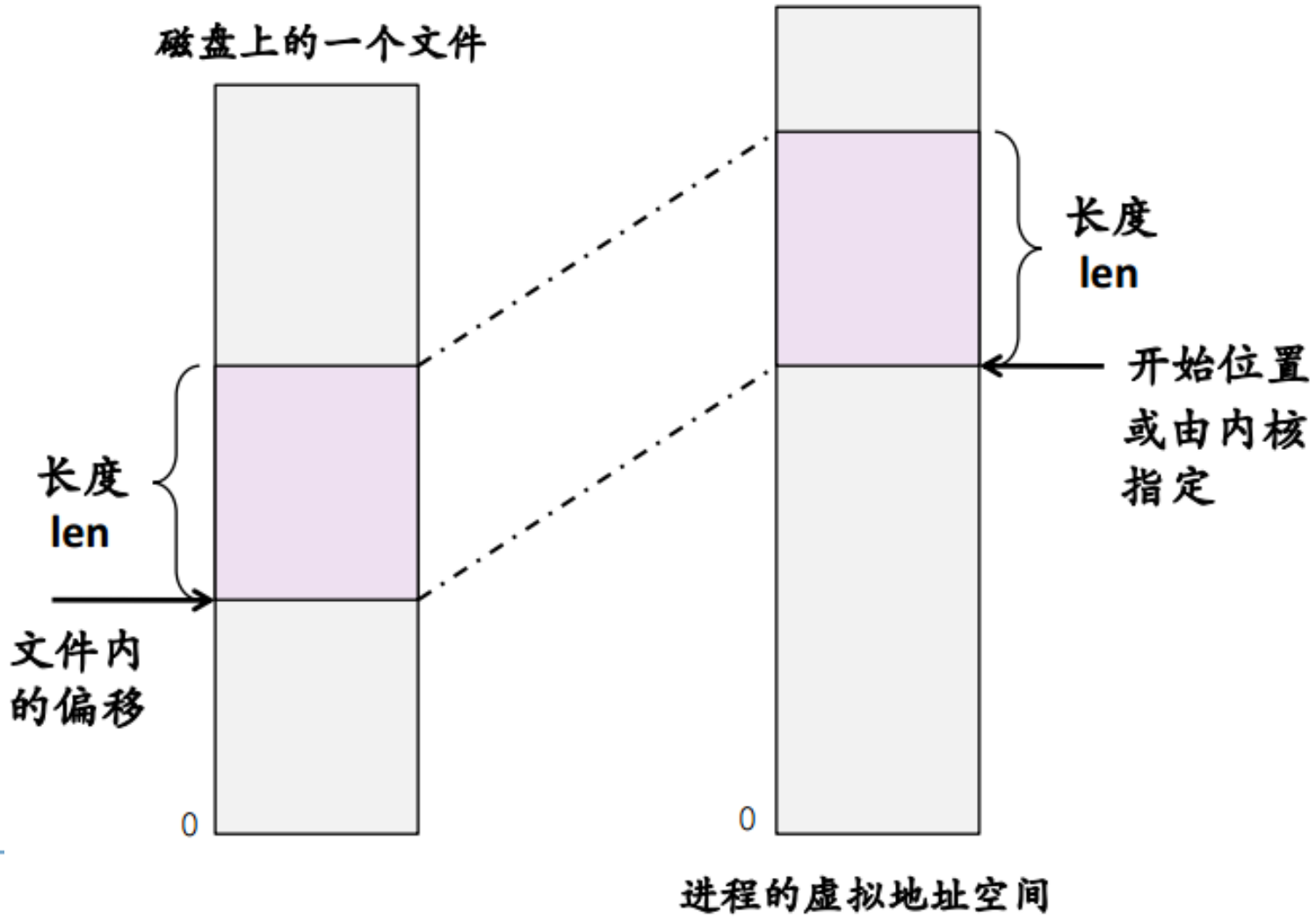
内存映射文件(Mem-Mapped File)

- 基本思想：将IO变成访存，简化读写操作，允许共享
  - 进程通过一个系统调用（mmap）将一个文件（或部分）映射到其虚拟地址空间的一部分，访问这个文件就像访问内存中的一个大数组，而不是对文件进行读写。
- 在多数实现中，在映射共享的页面时不会实际读入页面的内容，而是在访问页面时，页面才会被每次一页的读入，磁盘文件则被当作后备存储。
- 当进程退出或显式地解除文件映射时，所有被修改页面会写回文件。
- 采用内存映射方式，可方便地让多个进程共享一个文件。

Memory Mapped Files



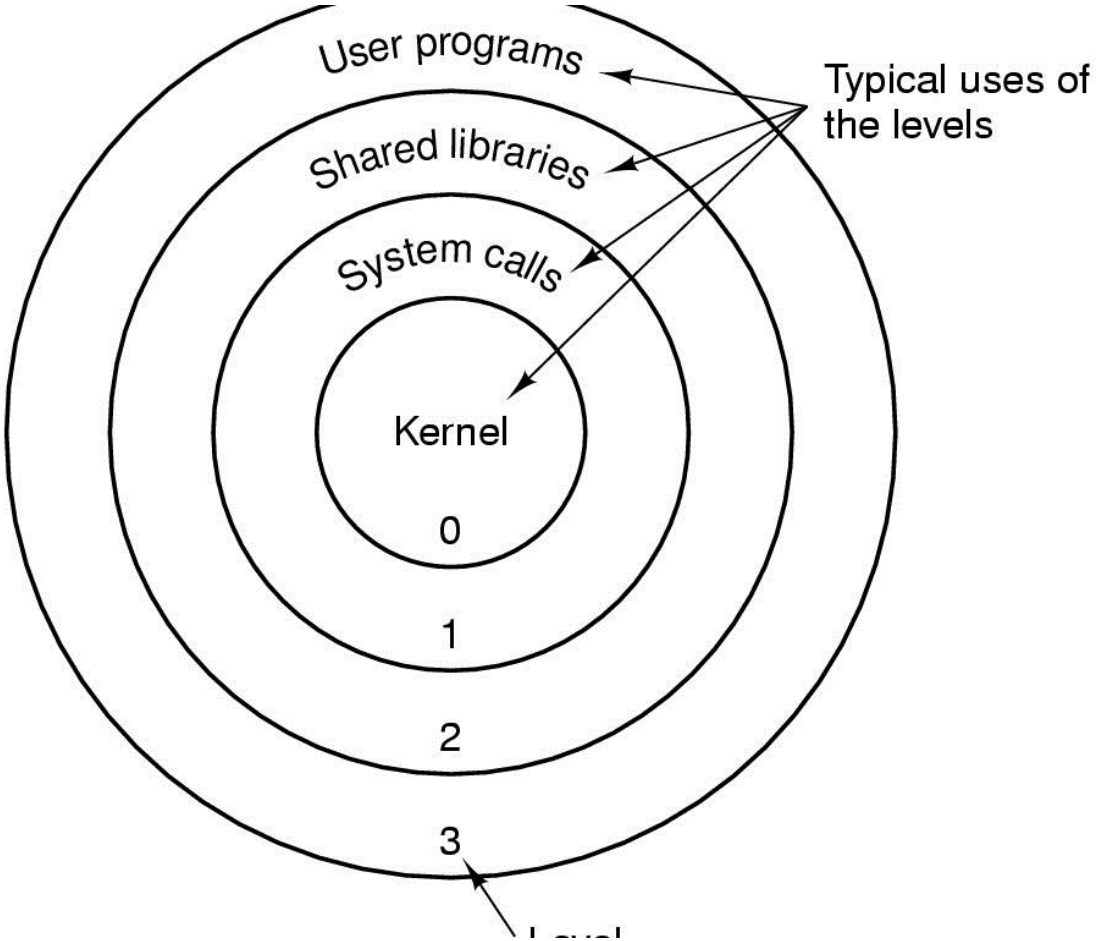
内存映射文件

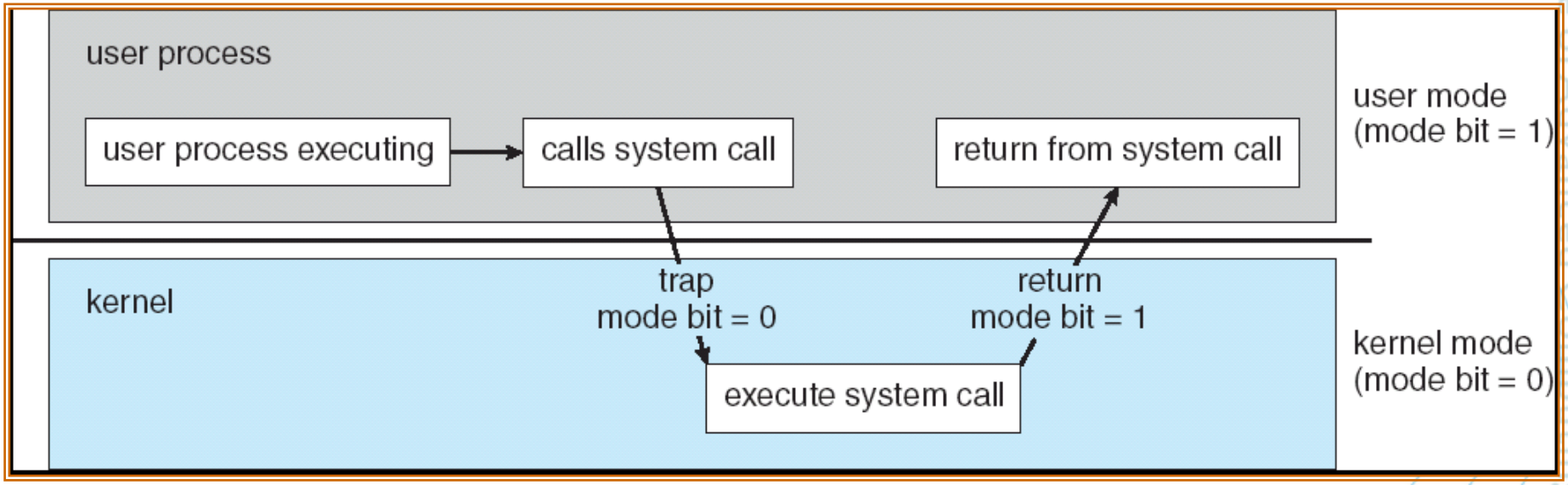


保护

- 界限保护（上下界界限地址寄存器）：所有访问地址必须在上下界之间；
- 用户态与内核态
- 存取控制检查
- 环保护：处理器状态分为多个环(ring)，分别具有不同的存储访问特权级别(privilege)，通常是级别高的在内环，编号小（如0环）级别最高；可访问同环或更低级别环的数据；可调用同环或更高级别环的服务。
  【权限的包含关系】

环保护





小结

- 存储组织（层次），存储管理功能
- 重定位和装入
- 静态链接和动态链接
- 存储管理方式：单一连续区管理，分区管理（静态和动态分区）
- 覆盖，交换
- 页式和段式存储管理：原理，优缺点，数据结构，地址变换，分段的意义，两者比较

虚拟存储器

- 局部性原理，虚拟存储器的原理
- 种类（虚拟页式、段式、段页式），缺页中断
- 存储保护和共享
- 调入策略、分配策略和清除策略
- 置换策略
- 工作集策略