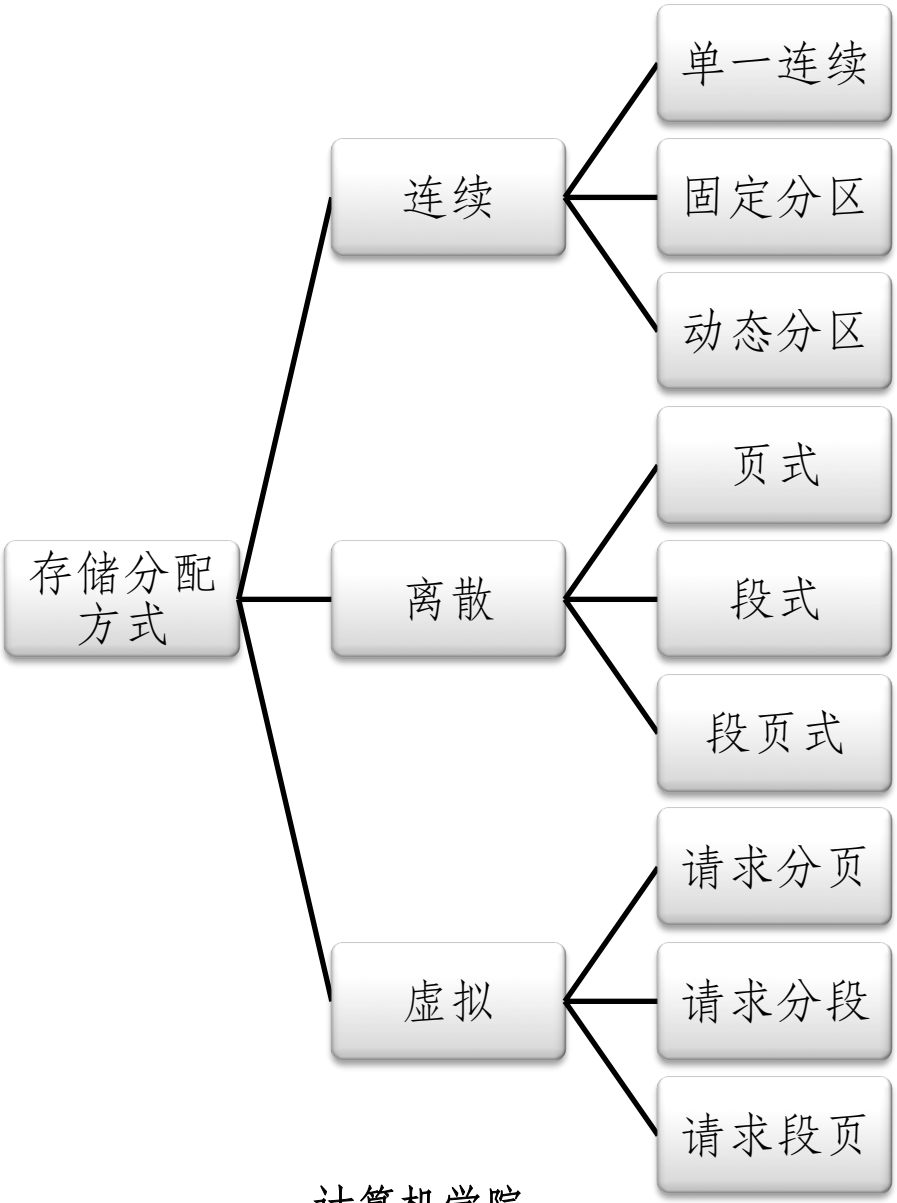
内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
- 存储管理实例

存储方式的分类



内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
  - 局部性原理
  - 请求式分页
  - 页面置换
  - 内存保护
- 存储管理实例

内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
  - 局部性原理
  - 请求式分页
  - 页面置换
  - 内存保护
- 存储管理实例

常规存储管理的问题

常规存储管理方式的特征：

- 一次性：要求一个作业全部装入内存后方能运行。
- 驻留性：作业装入内存后一直驻留内存，直至结束。

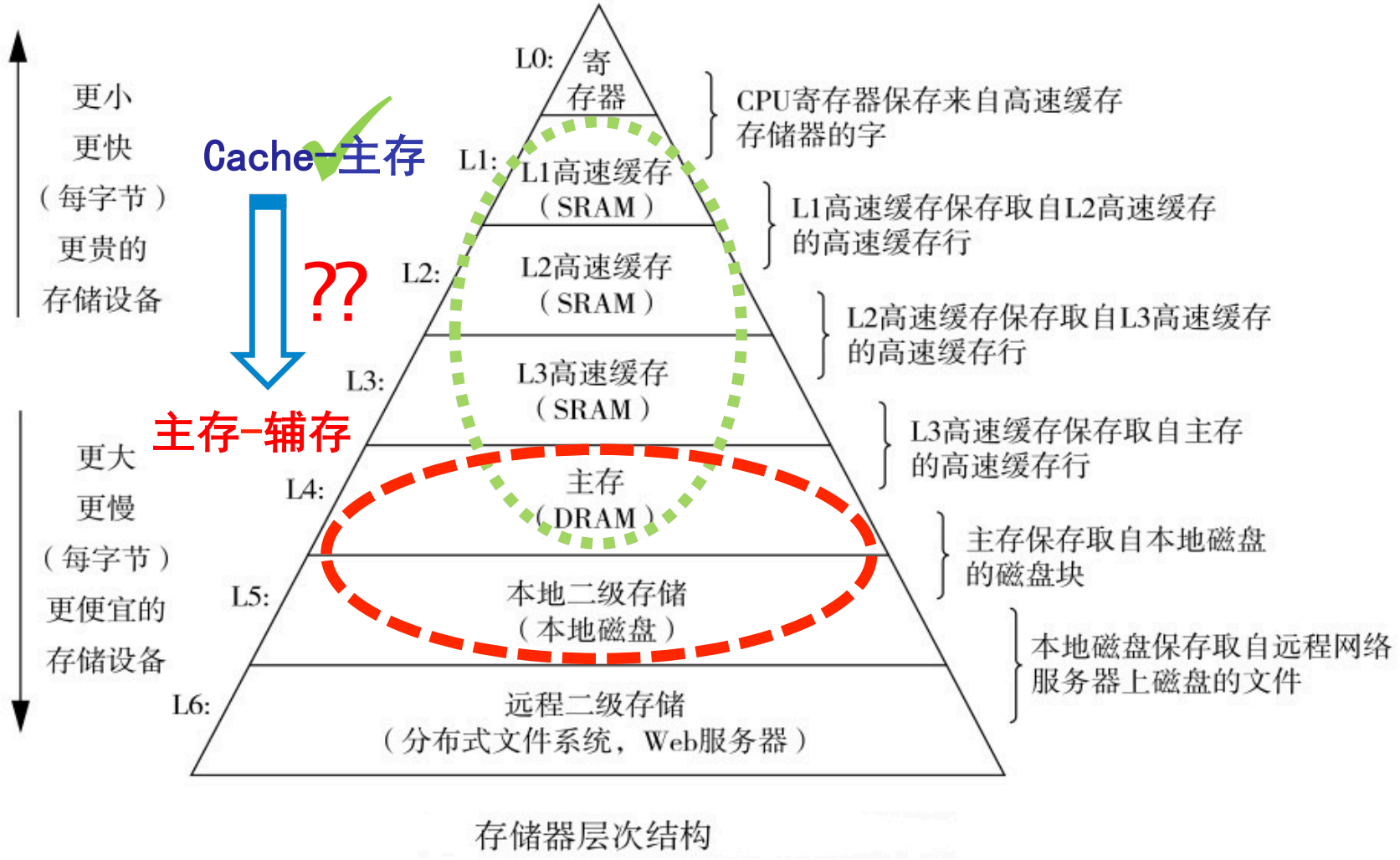
可能出现的问题：

- 有的作业很大，所需内存空间大于内存总容量，使作业无法运行。
- 有大量作业要求运行，但内存容量不足以容纳下所有作业，只能让一部分先运行，其它在外存等待。

解决方法：

- 增加内存容量
- 从逻辑上扩充内存容量：覆盖、对换。

再次回顾存储体系——借鉴已有方法



存储器层次结构

局部性原理

- 指程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。还可以表现为：
  - 时间局部性，即一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内；
  - 空间局部性，即当前指令和邻近的几条指令，当前访问的数据和邻近的数据都集中在一个较小区域内。

程序的局部性

- 程序在执行时，大部分是顺序执行的指令，少部分是转移和过程调用指令。（空间）
- 过程调用的嵌套深度一般不超过5，因此执行的范围不超过这组嵌套的过程。（空间）
- 程序中存在相当多的循环结构，它们由少量指令组成，而被多次执行。（时间）
- 程序中存在相当多对一定数据结构的操作，如数组操作，往往局限在较小范围内。（空间）

虚拟存储器的定义

定义：

虚拟存储器，是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。该系统逻辑容量由内存容量和外存容量之和所决定；其运行速度接近于内存速度，成本接近于外存。

虚拟存储器的基本原理

- 在程序装入时，不需要将其全部读入到内存，而只需将当前需要执行的部分页或段读入到内存，就可让程序开始执行。
- 在程序执行过程中，如果需执行的指令或访问的数据尚未在内存（称为缺页或缺段），则由处理器通知操作系统将相应的页或段调入到内存，然后继续执行程序。（请求调入功能）
- 另一方面，操作系统将内存中暂时不使用的页或段调出保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段。（置换功能）

虚拟存储技术的特征

- 离散性：物理内存分配的不连续，虚拟地址空间使用的不连续（数据段和栈段之间的空闲空间，共享段和动态链接库占用的空间）
- 多次性：作业被分成多次调入内存运行。正是由于多次性，虚拟存储器才具备了逻辑上扩大内存的功能。多次性是虚拟存储器最重要的特征，其它任何存储器不具备这个特征。
- 对换性：允许在作业运行过程中进行换进、换出。换进、换出可提高内存利用率。

虚拟存储技术的特征

- 虚拟性：通过物理内存和快速外存相结合，提供大范围的虚拟地址空间
  - 范围大，但占用容量不超过物理内存和外存交换区容量之和

虚拟性以多次性和对换性为基础，

多次性和对换性必须以离散分配为基础。

优点、代价和限制

优点：

- 可在较小的可用内存中执行较大的用户程序；
- 可在内存中容纳更多程序并发执行；
- 不必影响编程时的程序结构（与覆盖技术比较）
- 提供给用户可用的虚拟内存空间通常大于物理内存 (real memory)

代价：虚拟存储量的扩大是以牺牲 CPU 工作时间以及内外存交换时间为代价。

限制：虚拟存储器的容量取决于主存与辅存的容量，最大容量由计算机的地址结构决定。如 32 位机器，虚拟存储器的最大容量就是 4G，再大CPU 无法直接访问。

与Cache-主存机制的异同

相同点：

1. 出发点相同：二者都是为了提高存储系统的性能价格比而构造的分层存储体系，都力图使存储系统的性能接近高速存储器，而价格和容量接近低速存储器。

2. 原理相同：都是利用了程序运行时的局部性原理把最近常用的信息块从相对慢速而大容量的存储器调入相对高速而小容量的存储器。

与Cache-主存机制的异同

不同点：

1. 侧重点不同：cache主要解决主存与CPU的速度差异问题；虚存主要解决存储容量问题，另外还包括存储管理、主存分配和存储保护等方面。
2. 数据通路不同：CPU与cache和主存之间均有直接访问通路，cache不命中时可直接访问主存；而虚存所依赖的辅存与CPU之间不存在直接的数据通路，当主存不命中时只能通过调页解决，CPU最终还是要访问主存。

与Cache-主存机制的异同

3. 透明性不同：cache的管理完全由硬件完成，对系统程序员和应用程序员均透明；而虚存管理由软件（OS）和硬件共同完成，由于软件的介入，虚存对实现存储管理的系统程序员不透明，而只对应用程序员透明（段式和段页式管理对应用程序员“半透明”）。
4. 未命中时的损失不同：由于主存的存取时间是cache的存取时间的5～10倍，而主存的存取速度通常比辅存的存取速度快上千倍，故主存未命中时系统的性能损失要远大于cache未命中时的损失。

人类社会活动和生活的借鉴

- 这种管理模式在人类生活中十分常见：商品销售，家庭生活……

电脑配件销售方法	虚拟存储技术
配件存放于柜台与仓库，是一种存放体系	虚拟存储器由内存和外存共同组成，是一种存储体系
柜台取配件快，仓库取配件慢	内存访问速度快，外存访问速度慢
容量由柜台和仓库容量之和决定	容量由内存和外存容量之和决定
柜台单位容量成本高，仓库则低	内存单位容量价格高，外存则低
配件的销售也存在局部性现象	程序访问的局部性原理
柜台摆放顾客购买频率最高的配件	内存存放经常访问的数据
顾客要购买柜台没有而仓库有的配件，则出现缺货现象	存在缺页（段）现象
缺货时，需从仓库拿配件	缺页（段）时则需要请求调页（段），由缺页（段）中断机构调入
具有柜台和仓库之间货物的置换策略	有置换算法
以人力和租金便宜的仓库来换取租金昂贵的柜台空间	以CPU时间和外存空间换取宝贵的内存空间

虚存机制要解决的关键问题

1. 调度问题：决定哪些程序和数据应被调入主存。
2. 地址映射问题：在访问主存时把虚地址变为主存物理地址（这一过程称为内地址变换）；在访问辅存时把虚地址变成辅存的物理地址（这一过程称为外地址变换），以便存储块的交换。此外还要解决主存分配、存储保护与程序再定位等问题。
3. 替换问题：决定哪些程序和数据应被调出主存。
4. 更新问题：确保主存与辅存的一致性。

在操作系统的控制下，硬件和系统软件为用户解决了上述问题，从而使应用程序的编程大大简化。

实存管理与虚存管理

实存管理：

- 分区（Partitioning）（连续分配方式）（包括固定分区、可变分区）
- 分页（Paging）
- 分段（Segmentation）
- 段页式（Segmentation with paging）

虚存管理：

- 请求分页（Demand paging） – 主流技术
- 请求分段（Demand segmentation）
- 请求段页式（Demand SWP ）

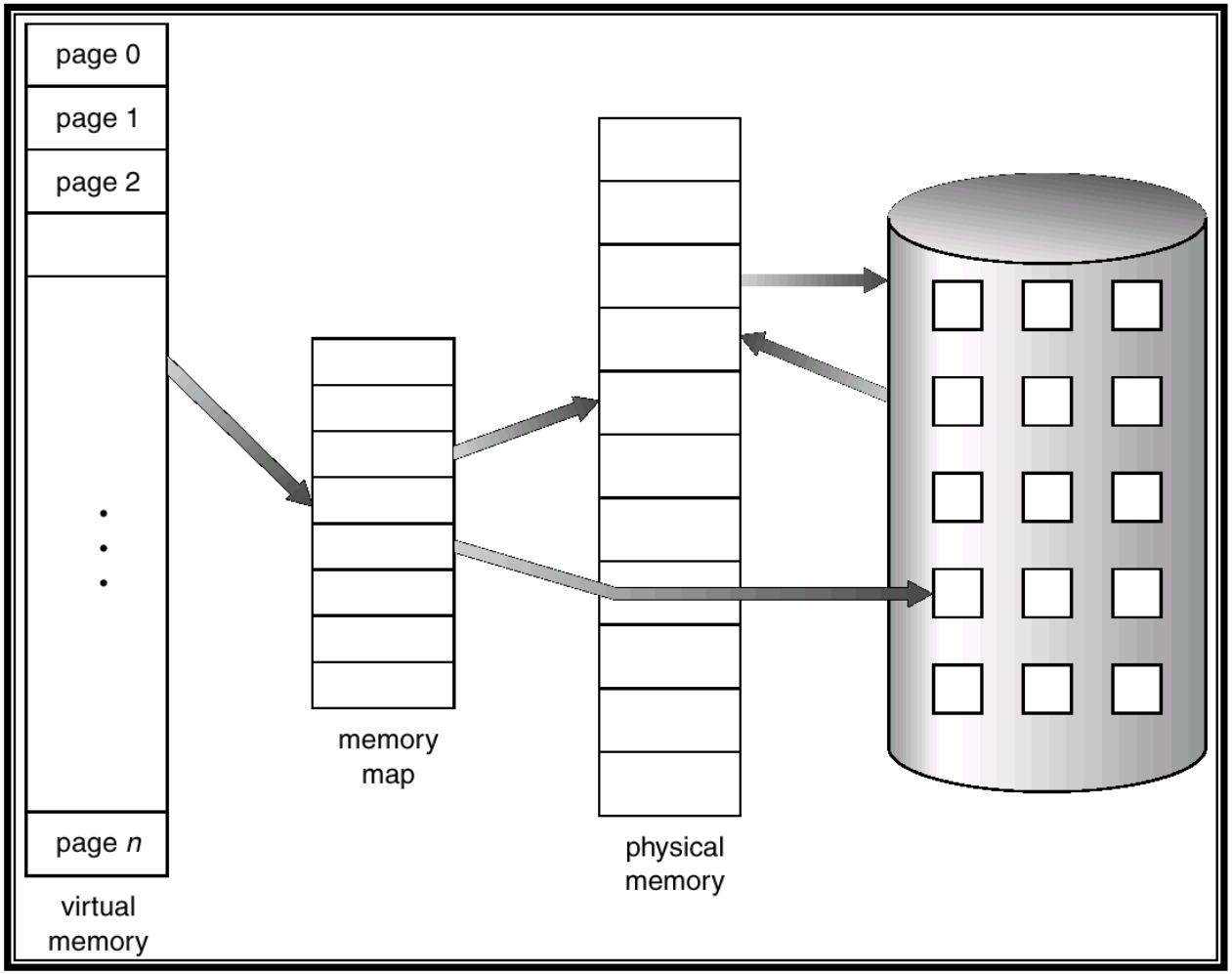
请求分页（段）系统

- 在分页(段)系统的基础上，增加了请求调页(段)功能、页面(段)置换功能所形成的页(段)式虚拟存储器系统。
- 它允许只装入若干页(段)的用户程序和数据，便可启动运行，以后在硬件支持下通过调页(段)功能和置换页(段)功能，陆续将要运行的页面(段)调入内存，同时把暂不运行的页面(段)换到外存上，置换时以页面(段)为单位。
- 系统须设置相应的硬件支持和软件：
  - 硬件支持：请求分页(段)的页(段)表机制、缺页(段)中断机构和地址变换机构。
  - 软件：请求调页(段)功能和页(段)置换功能的软件。

请求分页与分段系统的比较

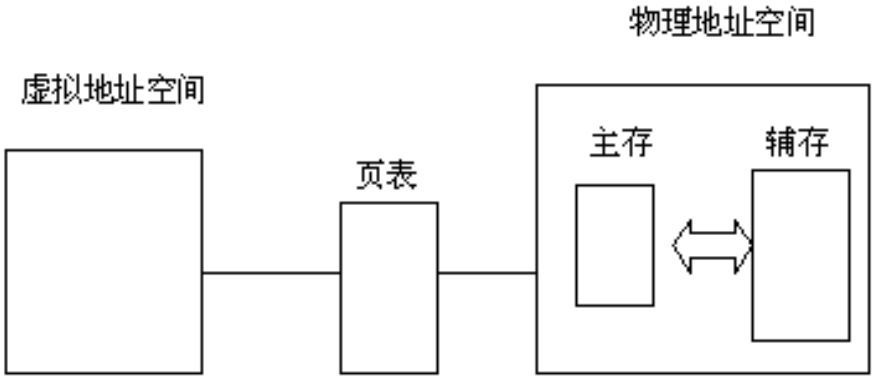
	请求分页系统	请求分段系统
基本单位	页	段
长度	固定	可变
分配方式	固定分配	可变分配
复杂度	较简单	较复杂

Virtual Memory That is Larger Than Physical Memory

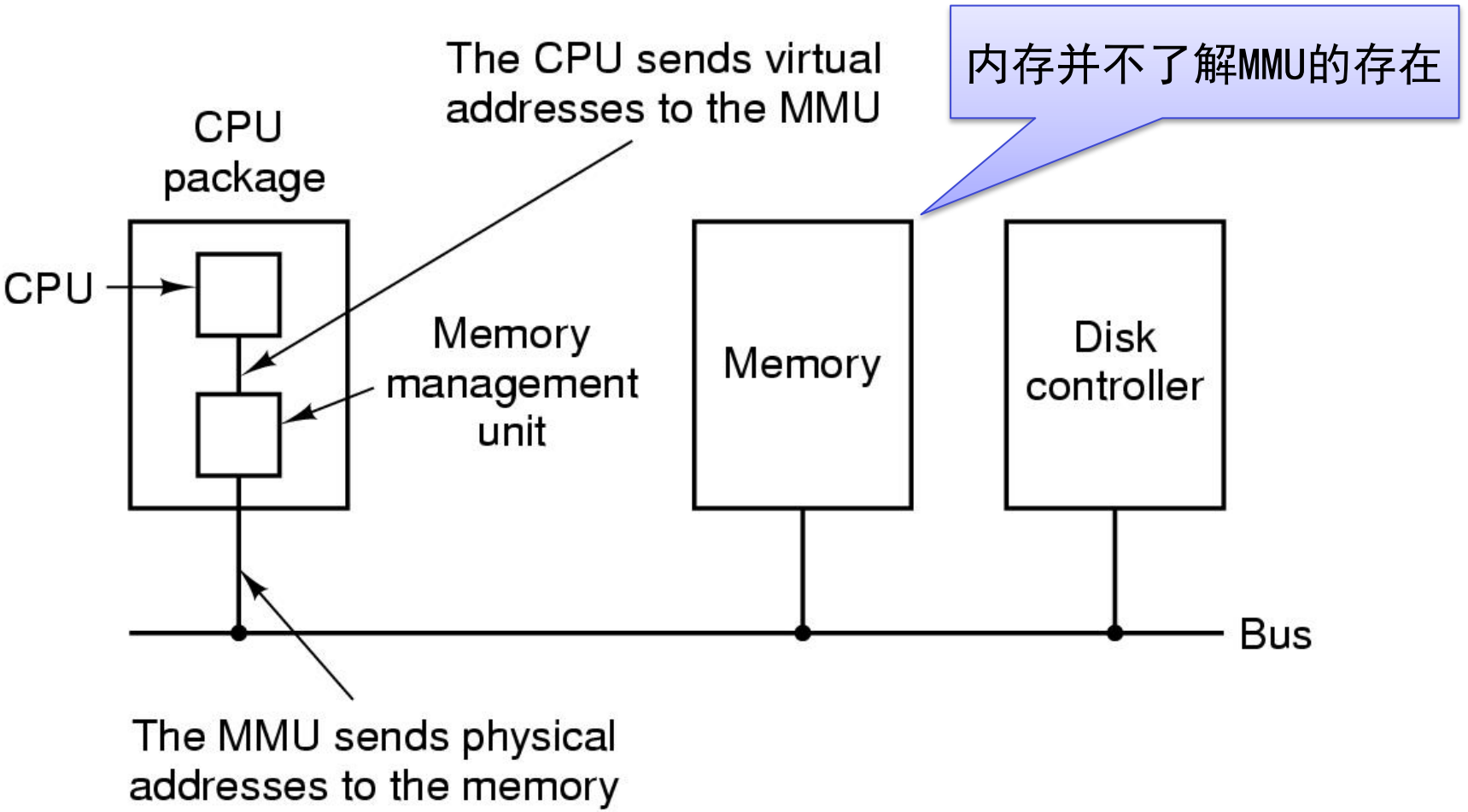


请求式分页系统

- 在运行作业之前，只要求把当前需要的一部分页面装入主存。当需要其它的页时，可自动的选择一些页交换倒辅存去，同时把所需的页调入主存。
- 虚拟存储系统：控制自动页面交换而用户作业意识不到的那个机构，成为虚拟存储系统。



MMU


内存并不了解MMU的存在
The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

页表机制

- 需要在进程页表中添加若干项
  - 状态位
  - 修改位(modified bit)
  - 外存地址
  - 访问字段
  - 禁用高速缓存

页表机制

- 状态位：
  - 用于指示该页是否已经调入了内存。
  - 一般由操作系统软件来管理，每当操作系统把一页调入物理内存中时，置位。相反，当操作系统把该页从物理内存调出时，复位。
  - CPU对内存进行引用时，根据该位判断要访问的页是否在内存中，若不在内存之中，则产生缺页中断

页表机制

- 修改位（modified 或者 脏位 dirty bit）：
  - 表示该页调入内存后是否被修改过。
  - 当CPU在写入一个页面时，硬件对该页的页表项中修改位置位。
  - 当OS重新分配页框的时候，如果页面已经被修改过（脏的），那么必须把它写回磁盘。如果没有被修改过（干净的），直接丢弃。

页表机制

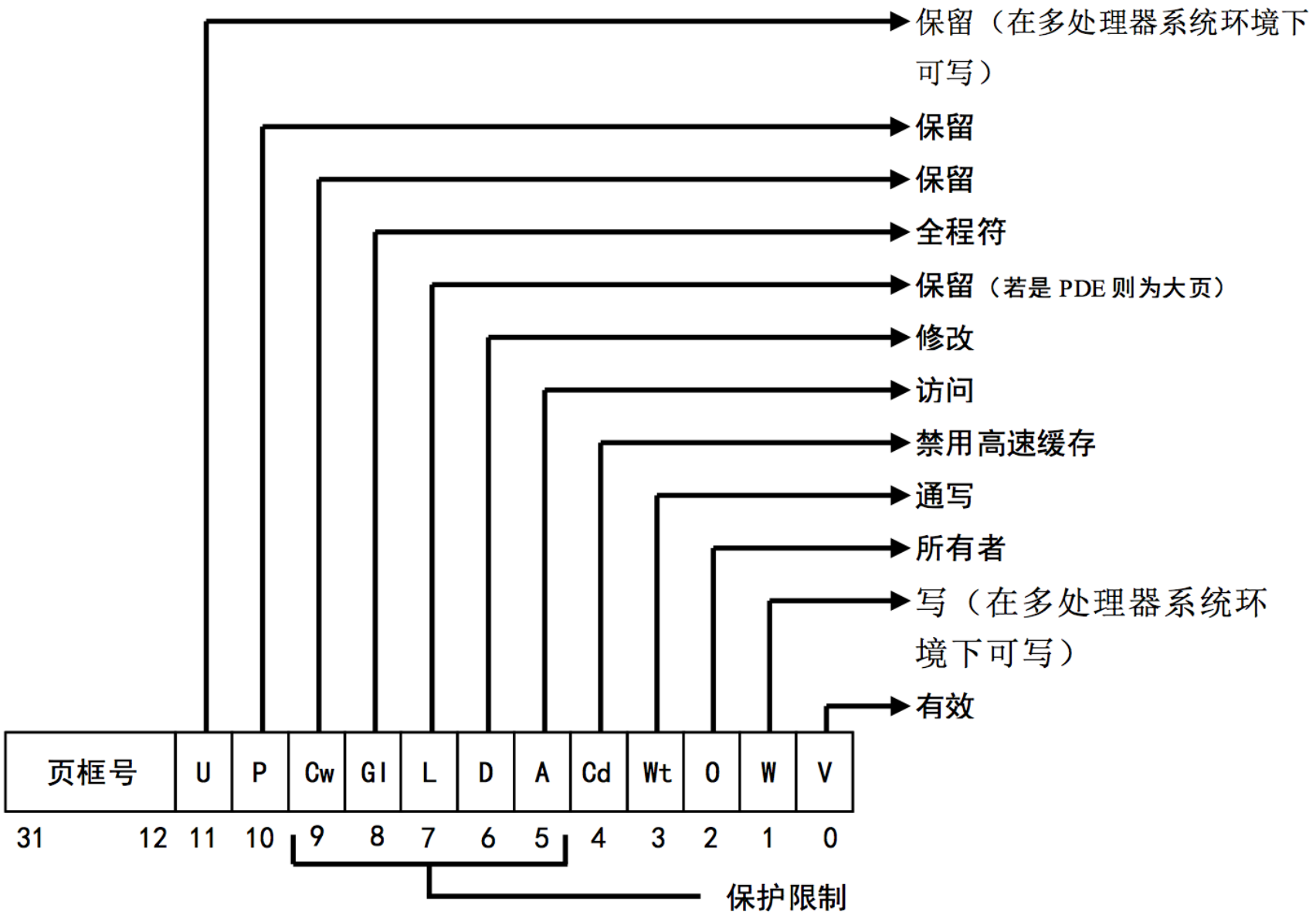
- 外存地址：
  - 用于指出该页在外存上的地址，供调入该页时使用
- 访问字段
  - 用于记录本页在一定时间内被访问的次数，或最近已经有多长时间未被访问。提供给相应的置换算法在选择换出页面时参考。

页表机制

- 禁用高速缓存
  - 适用于内存映射I/O
  - CPU通过循环等待查询外设对其命令的响应，保证从设备读取而不是从高速缓存中读取。

  - *内存映射，外设I/O端口的物理地址就被映射到CPU的单一物理地址空间中，而成为内存的一部分
  - *I/O映射，为外设专门实现了一个单独地地址空间，称为“I/O地址空间” 所有外设的I/O端口均在这一空间中进行编址。
    - CPU通过设立专门的I/O指令（如X86的IN和OUT指令）来访问这一空间中的地址单元（也即I/O端口）

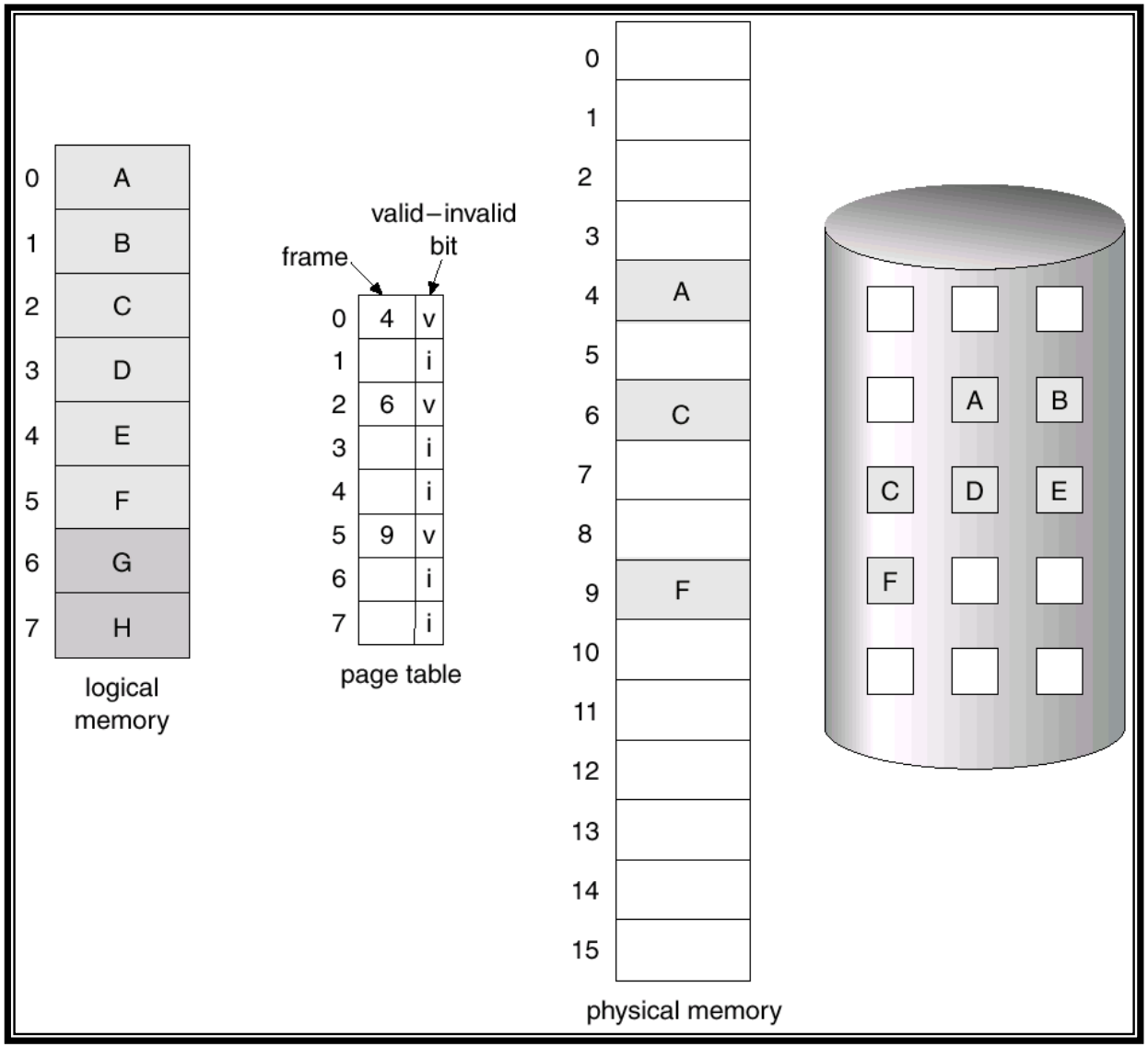
Intel x86页表项



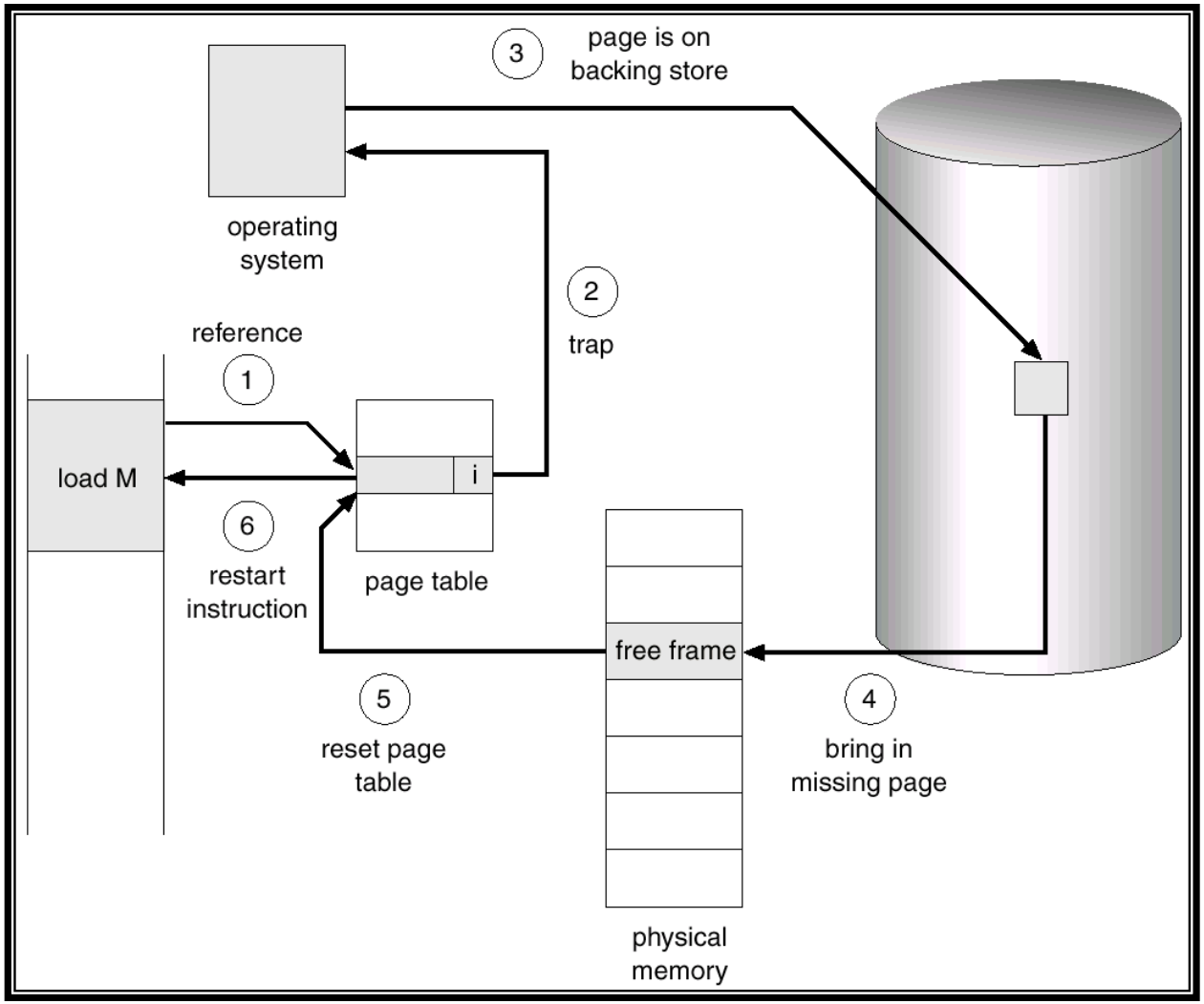
Intel x86页表项

标志位	含义
访问	此页已被读过。
禁用高速缓存	禁止对此页的高速缓存。
修改	此页已被写过。
全程符	变换对全部进程有效。（例如，变换缓冲区的刷新不会影响到这个页表项。）
大页	在有 128MB 内存以上系统中表示页目录项映射 4MB 的页面。（通常用于映射 Ntoskrnl 和 HAL、初始的非分页缓冲池等等。）
所有者	表明此页是否可以在用户态下访问，或仅可以在核心态下访问。
有效	表示变换是否映射到物理内存中实际的页面。
通写	写入此页时禁用高速缓存，这样数据的修改能立刻刷新外存上。
写	在单处理器系统上，表示此页是可读写的或只读的；在多处理器系统上，表示此页是否可写（这时写位存储在页表项保留位上。）。

地址变换


0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 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 A
5
6 C
7
8
9 F
10
11
12
13
14
15
physical memory
A B
C D E
F

缺页故障（Page Fault）处理



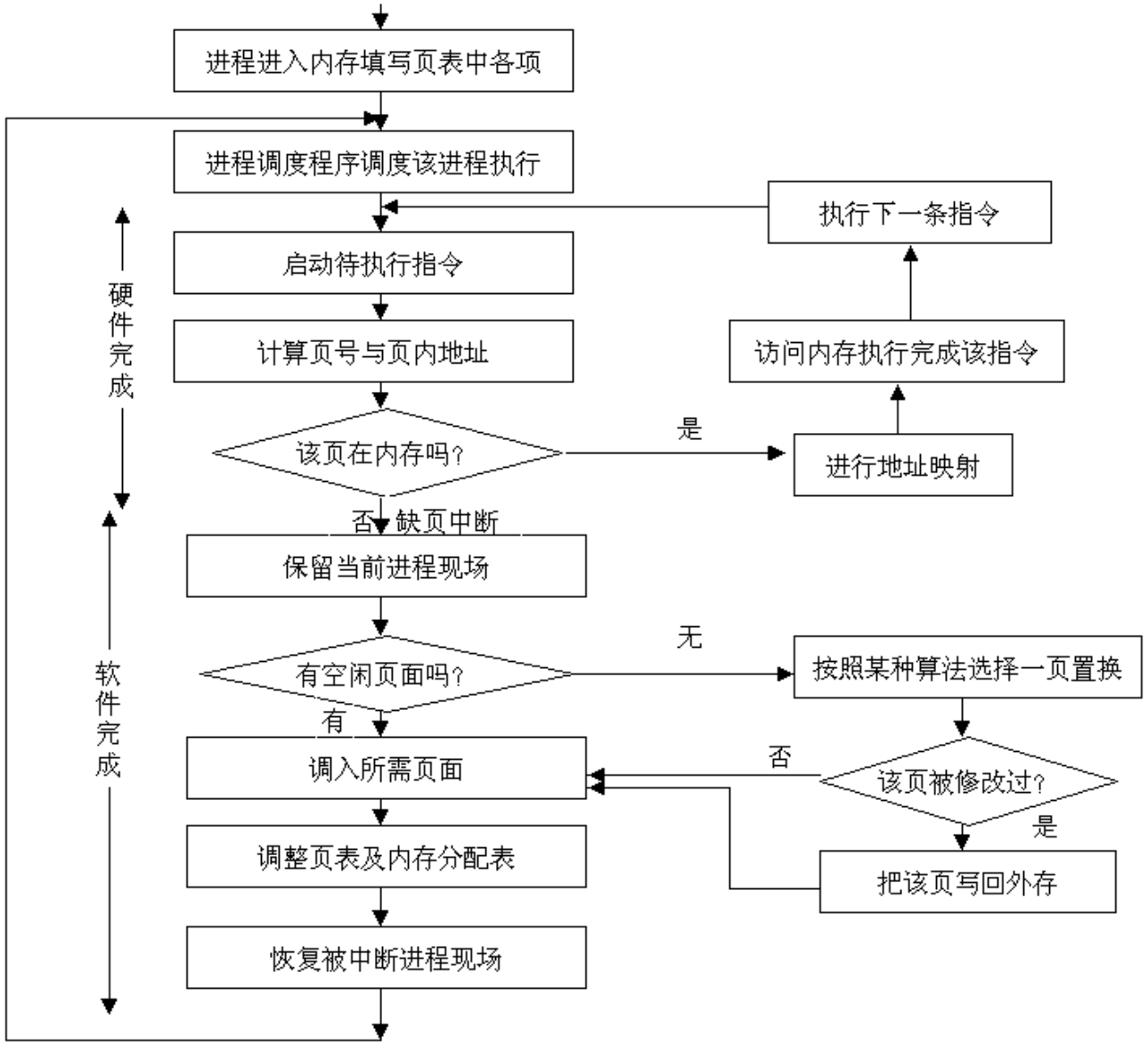
虚拟存储器的管理

- 置换问题
- 最小物理块数问题
- 分配问题

对缺页中断的支持

- 请求分页系统，CPU硬件需要支持缺页中断
  - 根据页表的状态位，确定是否产生缺页中断
- 缺页中断的特殊性：
  - 在指令的执行期间产生和处理缺页信号
    - 通常的CPU外部中断在当前指令执行完毕后检查
    - 缺页中断时指令执行期间发现指令或数据不在内存而产生
  - 一条指令可以产生多个缺页中断
    - 例如，多操作数的指令的操作数都不在内存，会产生多个中断
- 最终中断返回，还要能够正确执行产生中断的指令

对缺页中断的支持


进程进入内存填写页表中各项
进程调度程序调度该进程执行
启动待执行指令
计算页号与页内地址
该页在内存吗?
是
进行地址映射
访问内存执行完成该指令
执行下一条指令
否
缺页中断
保留当前进程现场
有空闲页面吗?
无
按照某种算法选择一页置换
该页被修改过?
否
是
把该页写回外存
有
调入所需页面
调整页表及内存分配表
恢复被中断进程现场
硬件完成
软件完成

页面调入策略

- 1. 请求调页（demand paging）
  - 只调入发生缺页时所需的页面。
  - 实现简单，但容易产生较多的缺页中断，造成对外存I/O次数多，时间开销过大，容易产生抖动现象。
- 2. 预调页（prepaging）
  - 在发生缺页需要调入某页时，一次调入该页以及相邻的几个页。
  - 这种策略提高了调页的I/O效率，减少了I/O次数。但由于这是一种基于局部性原理的预测，若调入的页在以后很少被访问，则造成浪费。
  - 常在程序装入时使用。

页面调入策略

- 调入页面的来源：
  - 1. 进程装入时，将其全部页面复制到交换区，以后总是从交换区调入。执行时调入速度快，要求交换区空间较大。
  - 2. 凡是未被修改的页面，都直接从文件区读入，而被置换时不需调出；已被修改的页面，被置换时需调出到交换区，以后从交换区调入。
    - 可能引发不一致的问题

页面置换策略（Replacement Strategies）

- 如果缺页中断发生时物理内存已满，“置换策略”被用于确定哪个虚页面必须从内存中移出为新的页面腾出空位。
- 在请求分页系统中，可采用两种分配策略，即固定和可变分配策略。
- 在进行置换时，也可以采用两种方式，即全局置换和局部置换。将它们组合起来有三种策略。

页面置换策略（Replacement Strategies）

- 固定分配局部置换（fixed allocation, local replacement）
  - 可基于进程的类型，为每一进程分配固定的页数的内存空间，在整个运行期间都不再改变。采用该策略时，如果进程在运行中出现缺页，则只能从该进程的N个页面中选出一个换出，然后再调入一页，以保证分配给该进程的内存空间不变。

页面置换策略（Replacement Strategies）

- 可变分配全局置换（variable allocation, global replacement）
  - 采用这种策略时，先为系统中的每一进程分配一定数量的物理块，操作系统本身也保持一个空闲物理块队列。当某进程发生缺页时，由系统的空闲物理块队列中取出一物理块分配给该进程。但当空闲物理块队列中的物理块用完时，操作系统才从内存中选择一块调出。该块可能是系统中任意一个进程的页。

页面置换策略（Replacement Strategies）

- 可变分配局部置换（variable allocation, local replacement）
  - 同样基于进程的类型，为每一进程分配一定数目的内存空间。但当某进程发生缺页时，只允许从该进程的页面中选出一页换出，这样就不影响其它进程的运行。如果进程在运行的过程中，频繁的发生缺页中断，则系统再为该进程分配若干物理块，直到进程的缺页率降低到适当程度为止。

分配和置换策略

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

初始页面配置

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

局部页面置换

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

全局页面置换

页面置换算法（Replacement Algorithm）

- 置换算法（replacement algorithm）决定在需要调入页面时，选择内存中哪个物理页面被置换。
- 置换算法的出发点应该是，把未来不再使用的或短期内较少使用的页面调出，而未来的实际情况是不确定的…
- 在局部性原理指导下依据过去的统计数据进行预测。
- 目的：是把那些访问概率非常高的页放入内存，减少内外存交换的次数。

页面置换算法（Replacement Algorithm）

- 页面置换问题在其他领域也会产生：
  - 高速缓存（Cache）
  - Web服务器：经常访问的Web页面

最佳算法（OPT optimal）

- 选择“未来不再使用的”或“在离当前最远位置上出现的”页面被置换。
  - 这是一种理想情况，是实际执行中无法预知的，因而不能实现，只能用作性能评价的依据。

最近未使用页面置换算法NRU

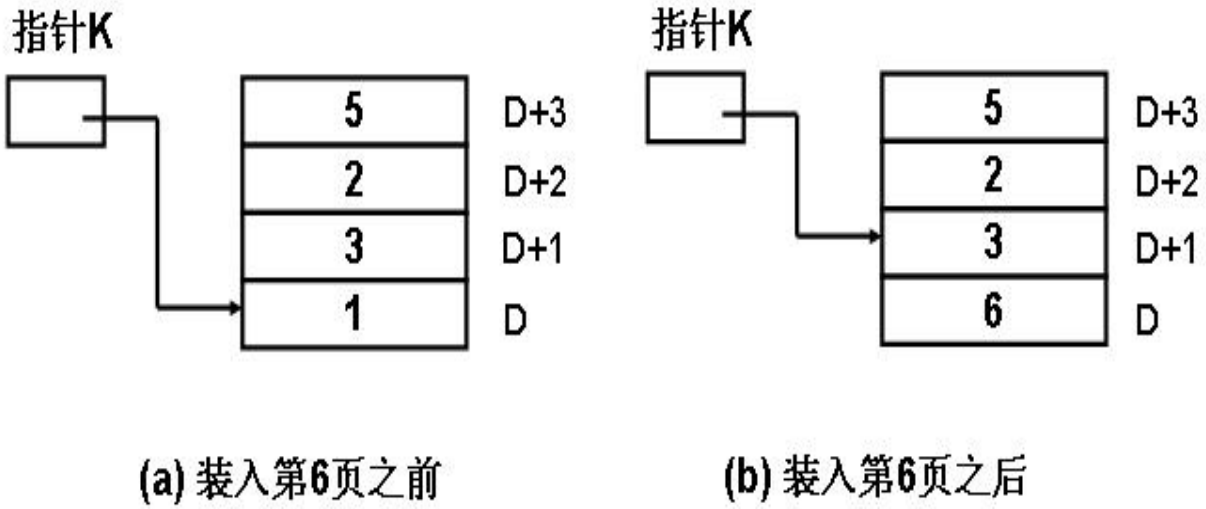
- 在最近一个时钟滴答内，套题一个没有被访问的已修改页面比淘汰一个频繁使用的干净页面摇号。
- 实现方式
  - 硬件记录引用位R和修改位M
  - 时钟定期清除R
  - Class 0 R=0, M=0
  - Class 1 R=0, M=1
  - Class 2 R=1, M=0
  - Class 3 R=1, M=1
  - 从编号最小的非空类中挑选一个淘汰

先进先出算法（FIFO）

- FIFO（First In First Out）
- 总是选择最先装入内存的一页调出，或者说是把驻留在内存中时间最长的一页调出.
  - 算法容易实现。把装入内存的页面的页号按进入的先后次序排好队列，每次总是调出队首的页，当装入一个新页后，把新页的页号排入队尾
  - 可以通过链表实现。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。

先进先出算法（FIFO）

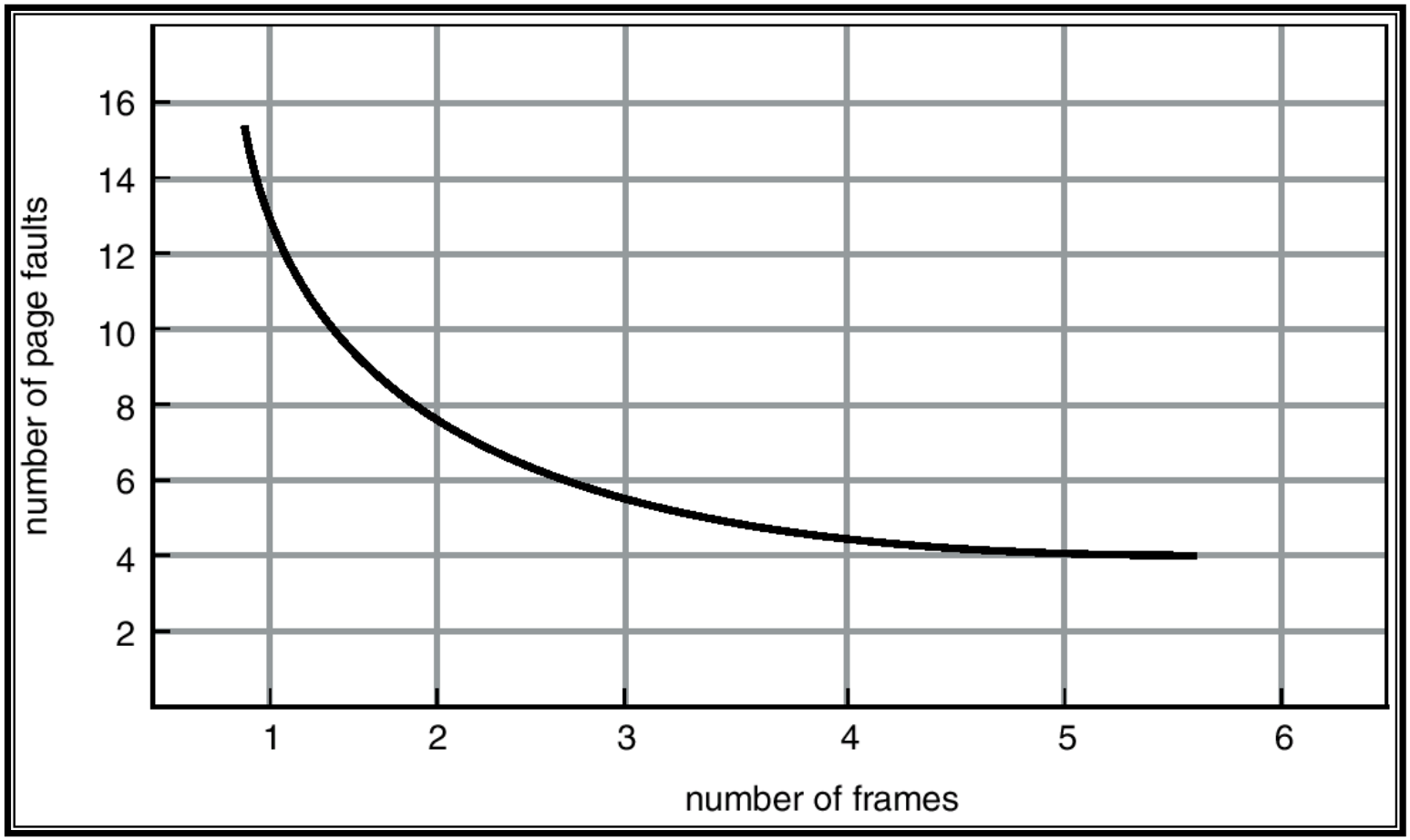
- 用指针K指示最早被装入内存的那页在队列中的位置，每次总是选择指针K指示的页调出，在装入一个新页后，在指针指示的位置上填上新页页号，然后指针K加1，指出下一次可调出的页 。
- 这里指针K是循环指针，假定页号队列中有n个页号，每次调出一页后，执行 K：＝（K＋1）mod n
- K就成为队首，新加入页成为队尾。



(a) 装入第6页之前　　(b) 装入第6页之后

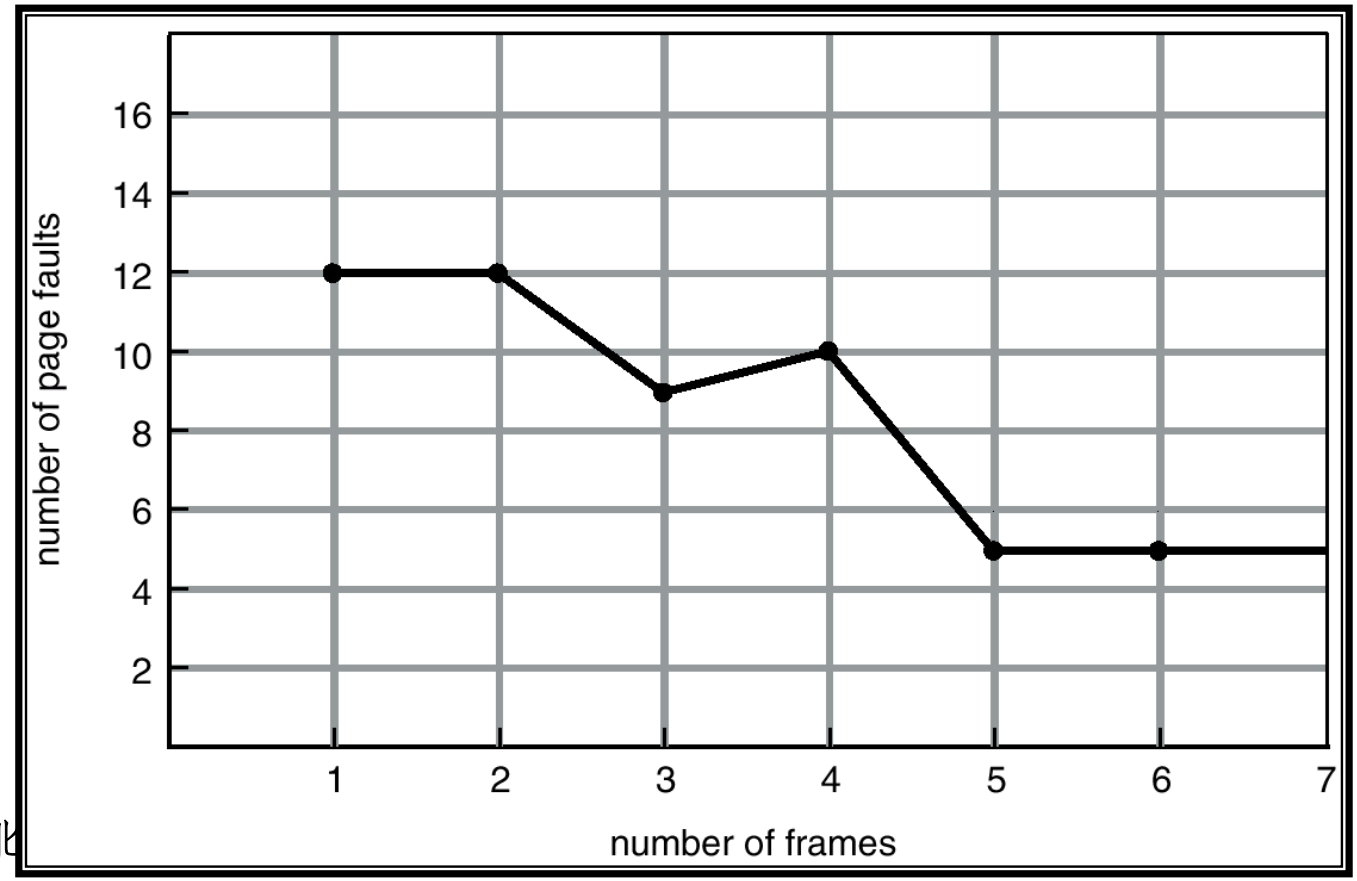
FIFO的Belady现象

- 理想的情况：缺页率随页框数增加而下降



FIFO的Belady现象

- Belady：采用FIFO算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多，缺页率反而提高的异常现象。
- Belady于1969年发现，故称为Belady现象



FIFO的Belady现象

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	3	2	4	4	4	1	0	0
		3	2	1	0	3	2	2	2	4	1	1
Oldest Page			3	2	1	0	3	3	3	2	4	4

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	0	0	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1
Oldest Page				3	3	3	2	1	0	4	3	2

3个页框9次缺页，4个页框10次缺页(红色)

二次机会页面置换算法

- 对FIFO的改进：
- 检查最老页面的R位，如果R=0，老且没有被用，淘汰
- 如果R=1，把R清零后，放在链表最后，修改装入时间。

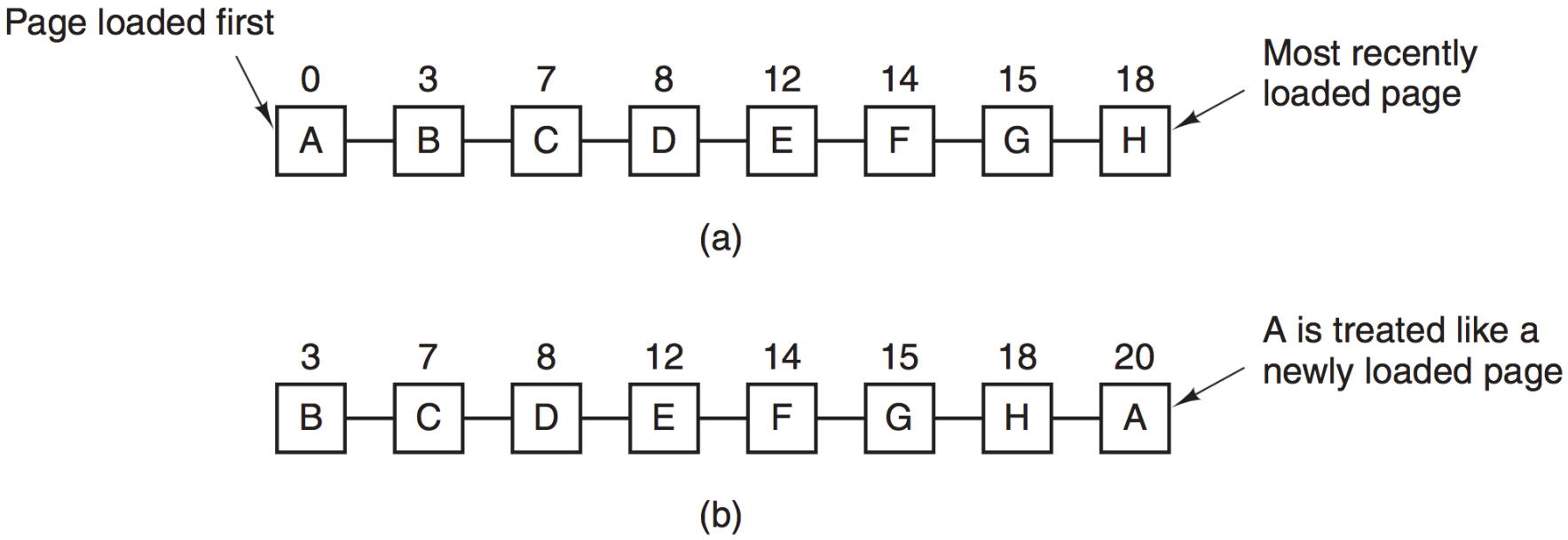


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

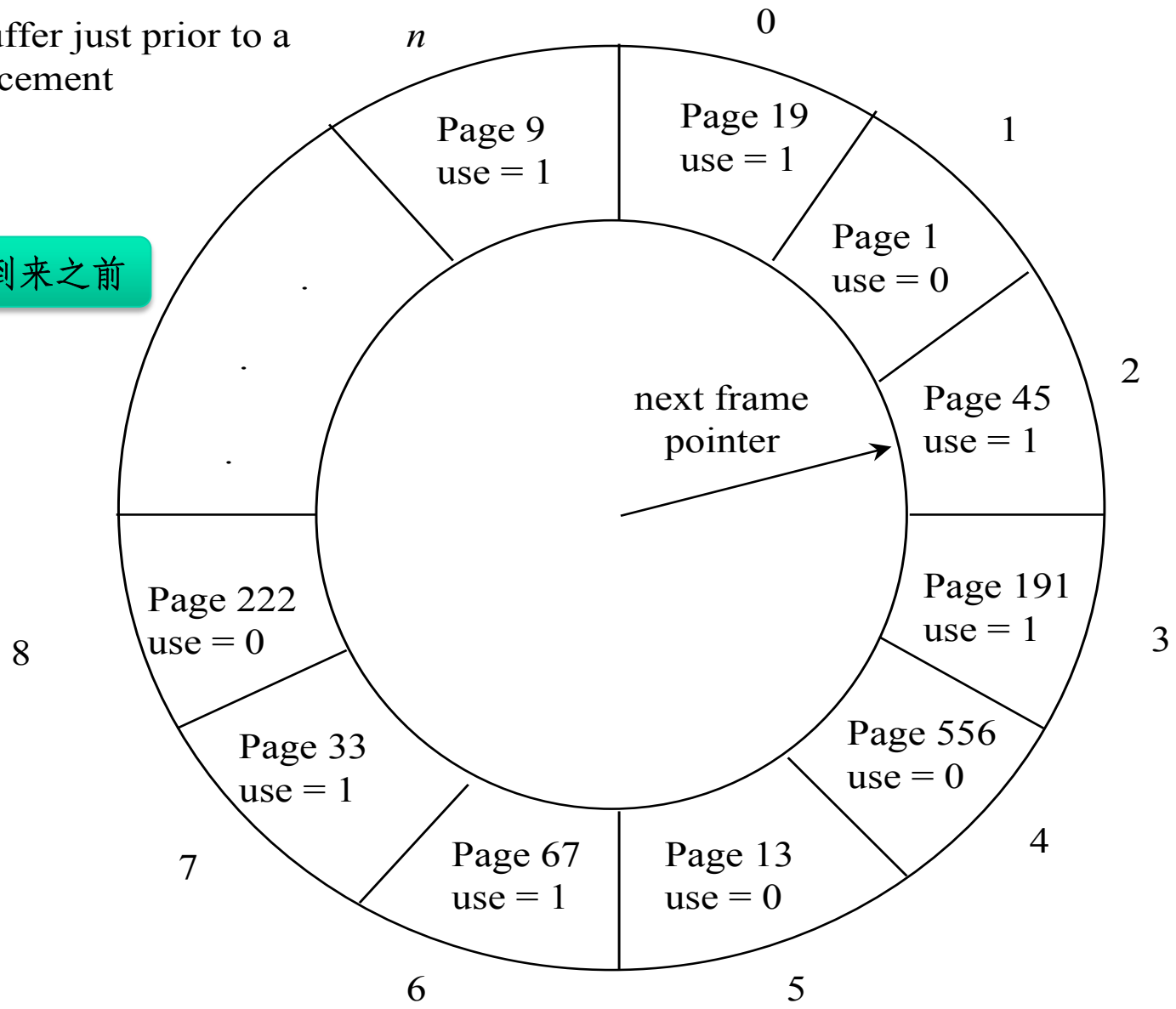
时钟算法Clock

- 时钟算法
  - 每页有一个使用标志位(use bit)，若该页被访问则置 use bit=1。(硬件完成)
  - 置换时采用一个指针，从当前指针位置开始按地址先后检查各页，寻找use bit=0的页面作为被置换页。
  - 指针经过的use bit=1的页都修改use bit=0，最后指针停留在被置换页的下一个页。(OS完成)

时钟算法Clock

State of buffer just prior to a page replacement

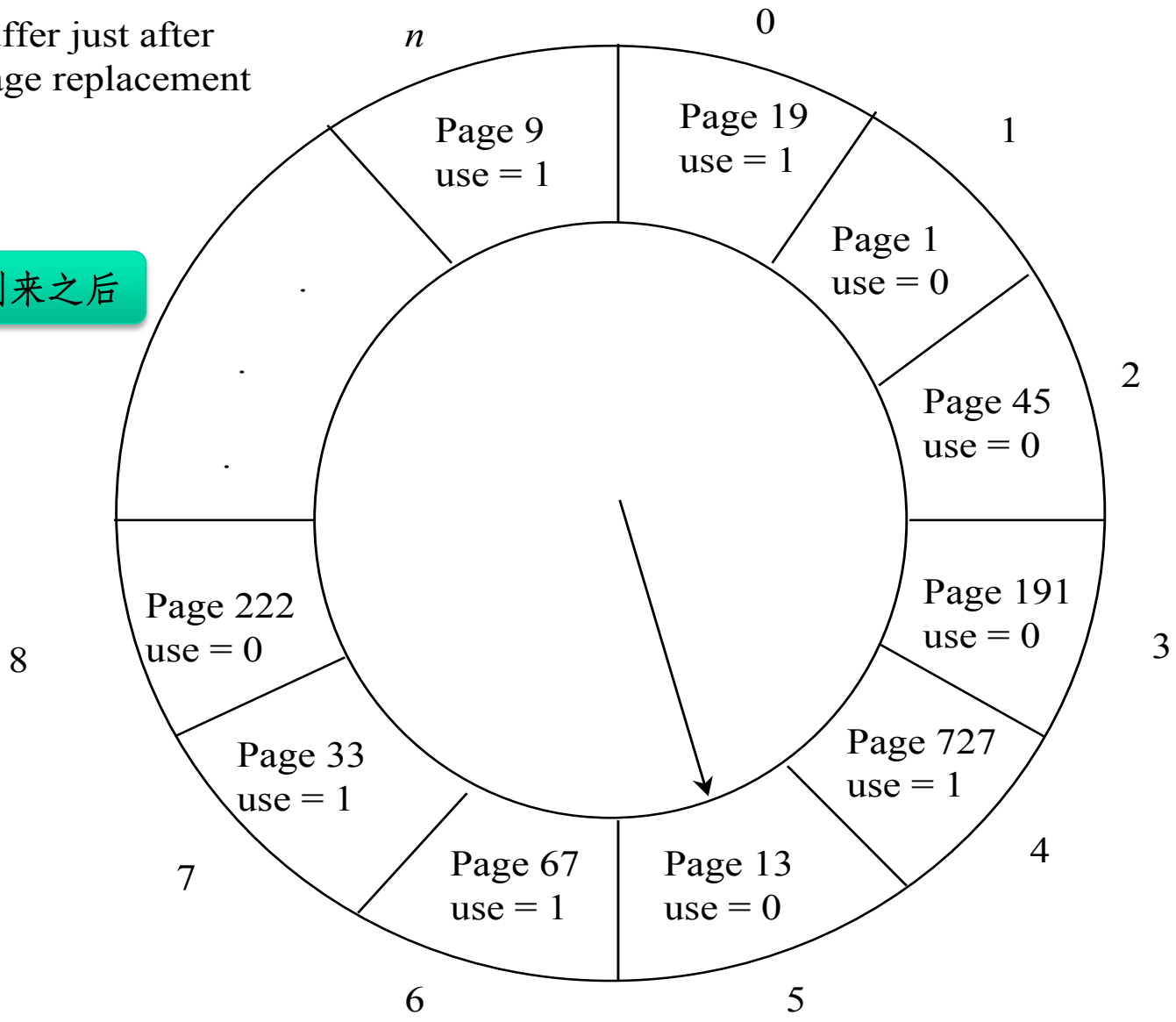
Page 727 到来之前



时钟算法Clock

State of buffer just after
the next page replacement

Page 727 到来之后



最近最久不用的页面置换算法

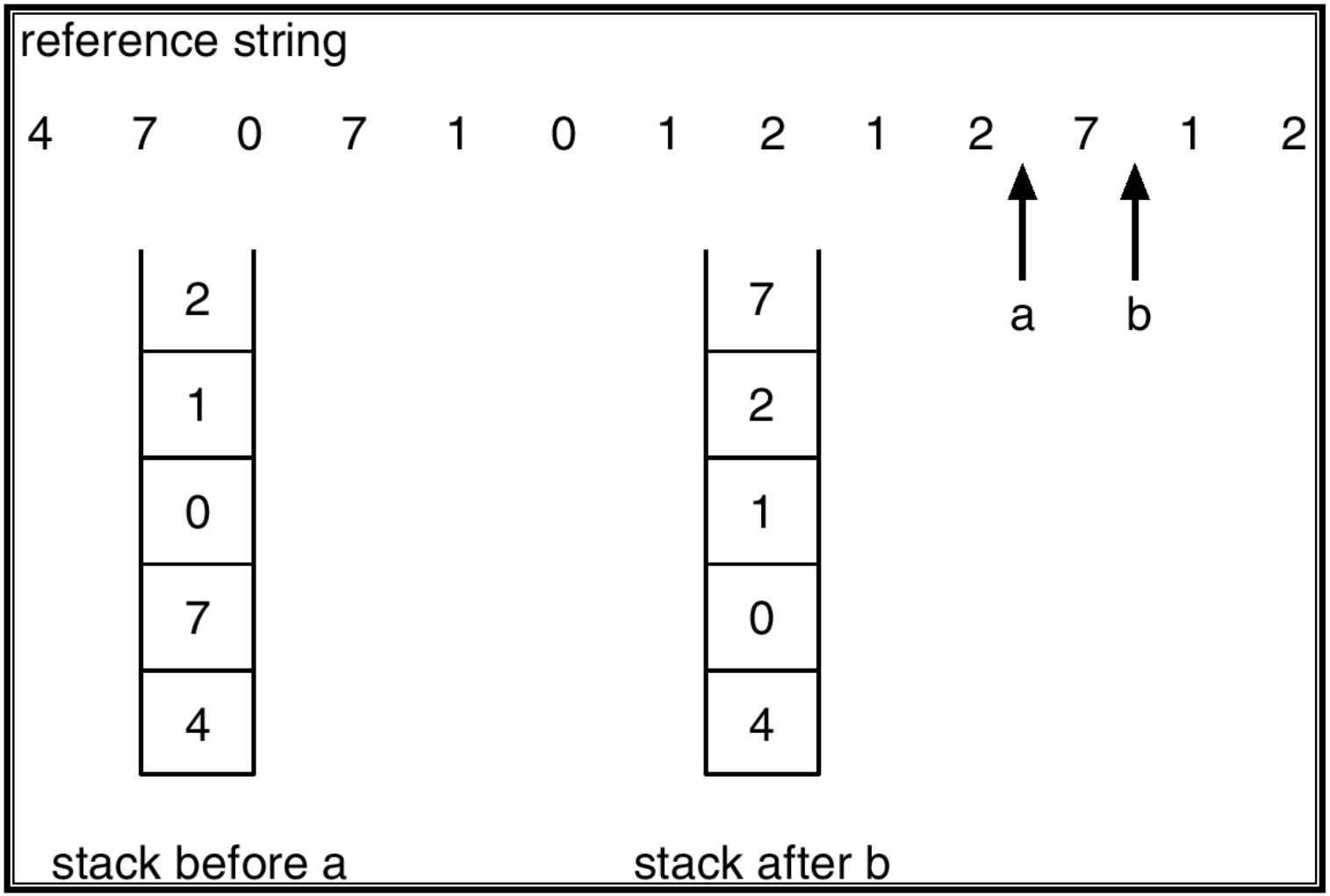
- LRU（Least Recently Used Replacement）
- 当需要置换一页面时，选择在最近一段时间内最久不用的页面予以淘汰。
  - 依据局部性原理
  - 前面几条指令中使用频繁的页面很可能在后面的几条指令中也频繁使用。
  - 反过来说，已经很久没有使用的页面很有可能在未来较长的一段时间内不会被用到

最近最久不用的页面置换算法

- LRU（Least Recently Used Replacement）
  - 性能接近最佳算法
  - 要记录页面使用时间的先后关系，硬件开销大
    - 使用特殊的堆栈：把被访问的页面移到栈顶，于是栈底的是最久未使用页面 。
    - 每个页面设立移位寄存器：被访问时左边最高位置1，定期右移并且最高位补0，于是寄存器数值最小的是最久未使用页面

最近最久不用的页面置换算法

- 最近访问的页面放在栈顶

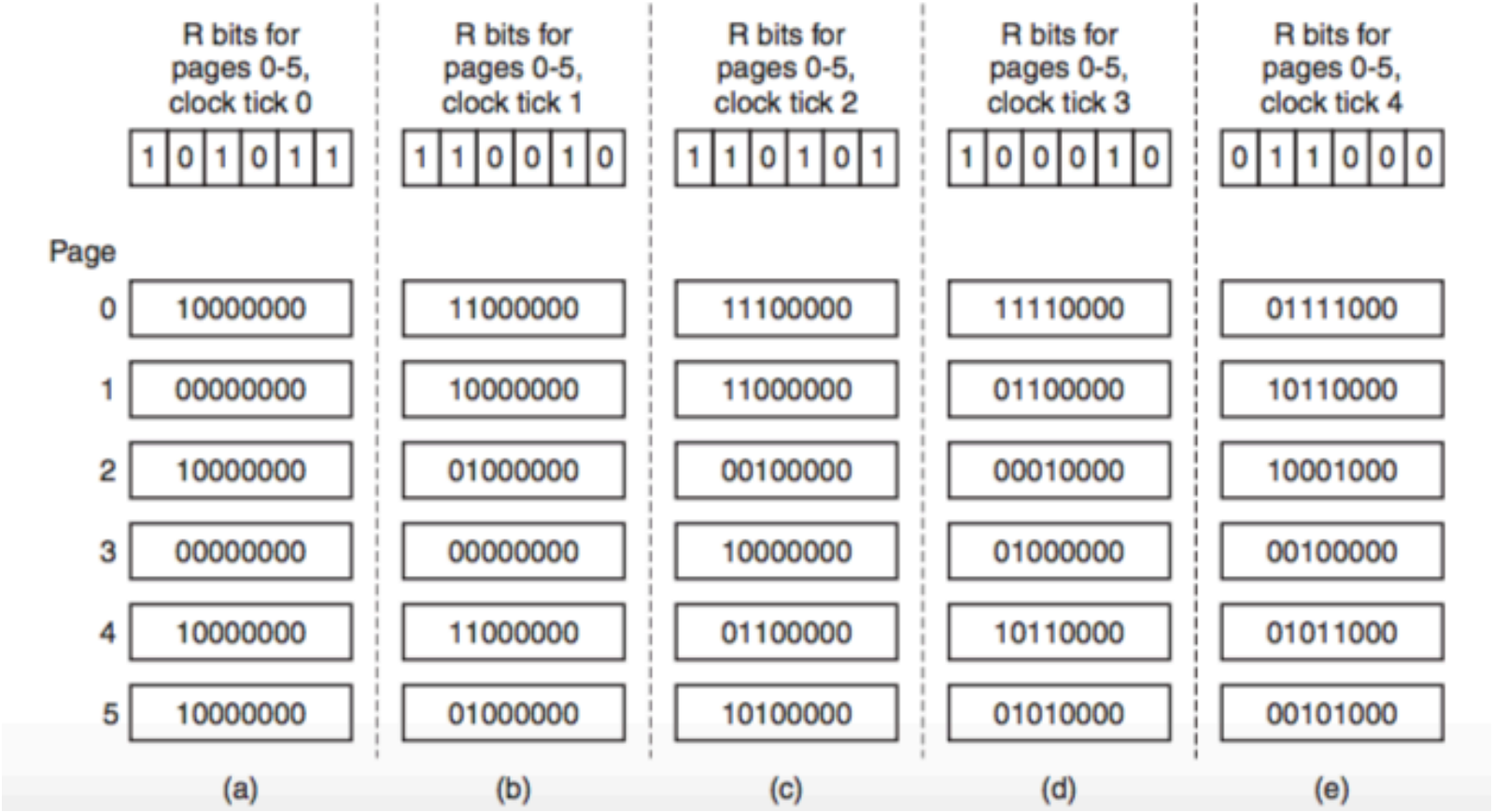


最不常用算法(NFU, Not Frequently Used)

- 对LRU的近似实现
- 选择到当前时间为止被访问次数最少的页面被置换;
- 每页设置访问计数器，每当页面被访问时，该页面的访问计数器加1;
- 发生缺页中断时，淘汰计数值最小的页面，并将所有计数清零;
- 问题：NFU从不忘记

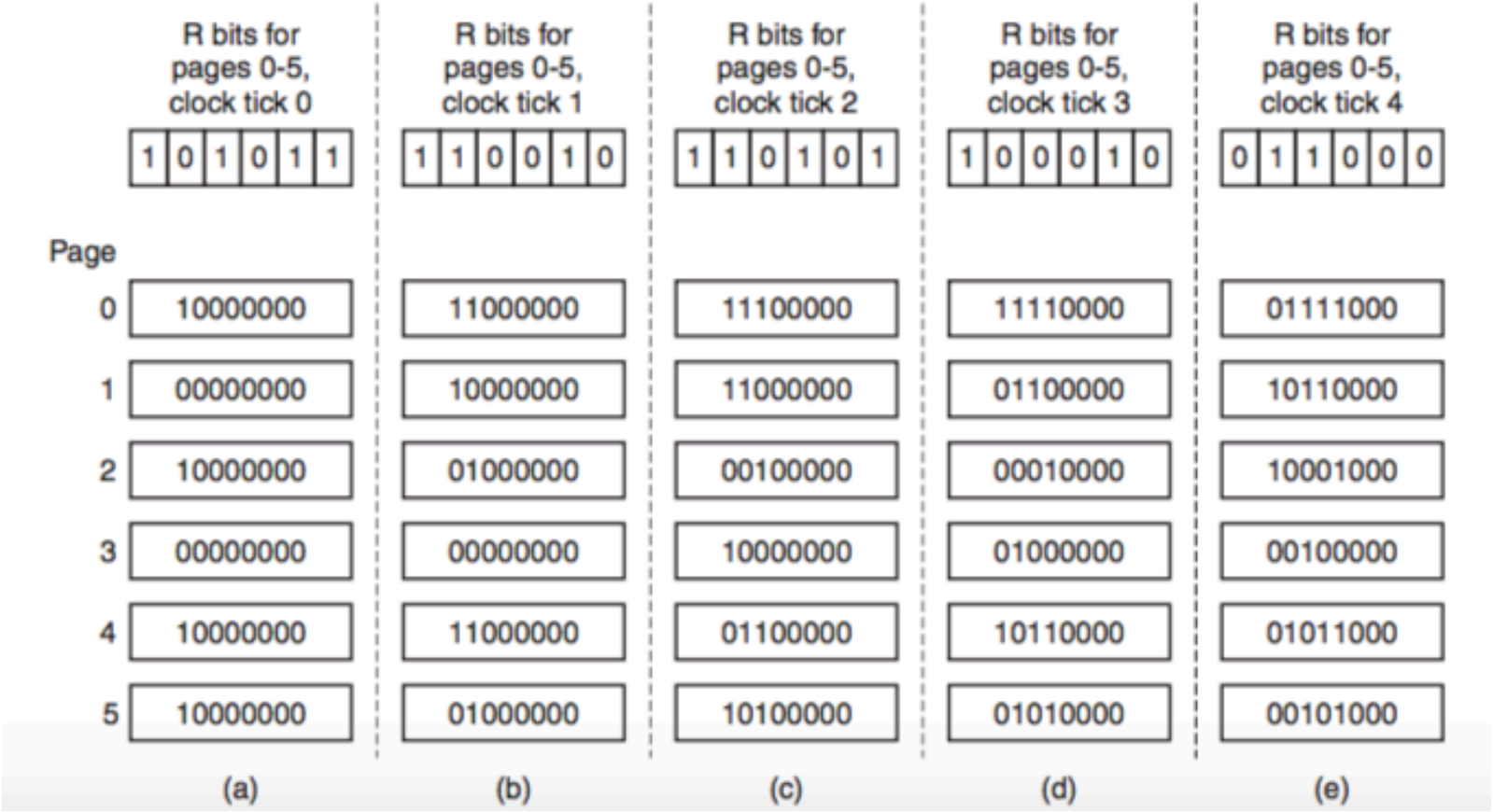
Aging算法

- 对NRU改进，每次计数器右移一位，R位添加到最左边



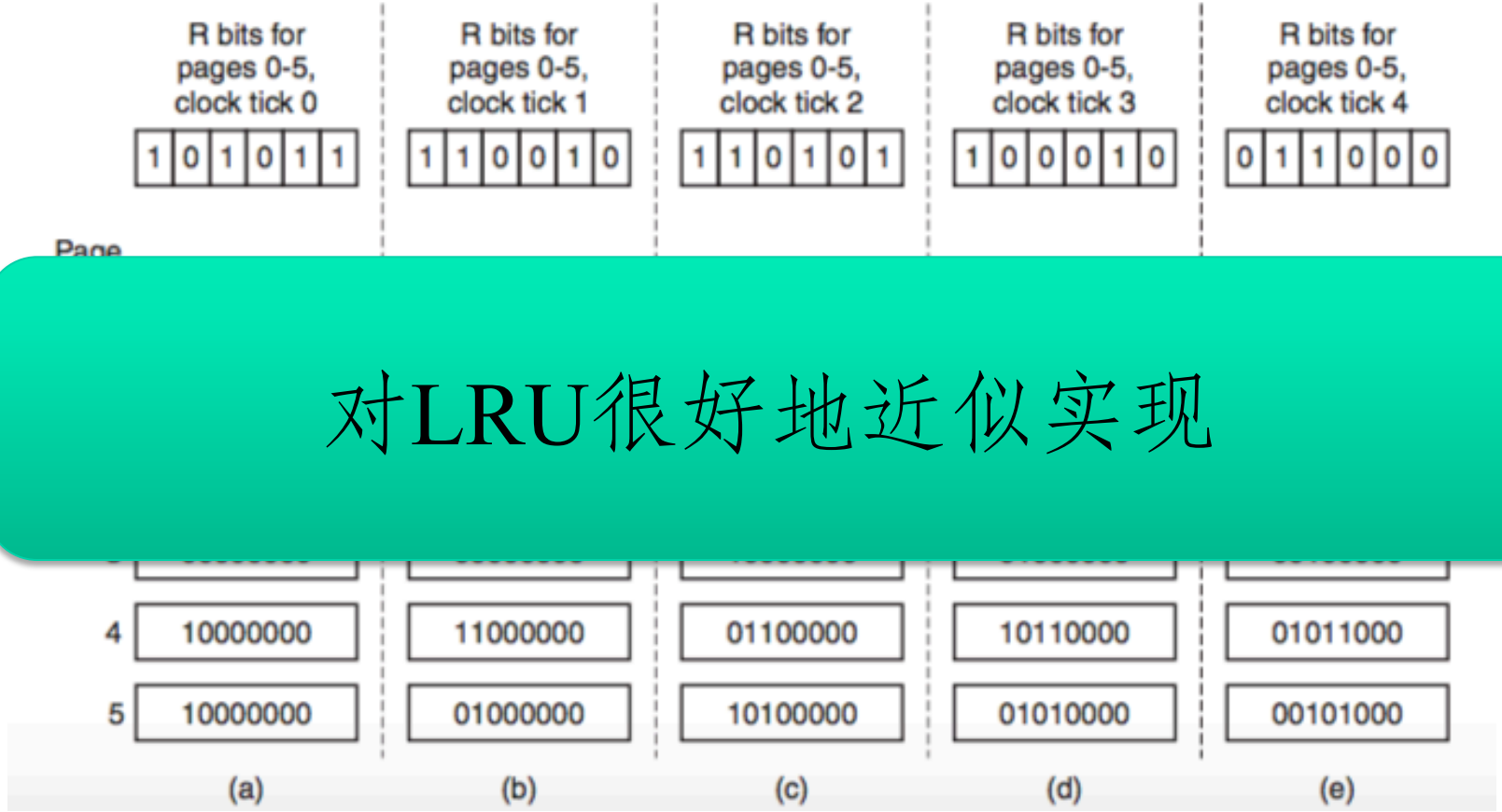
Aging算法

- 对NRU改进，每次计数器右移一位，R位添加到最左边



Aging算法

- 对NRU改进，每次计数器右移一位，R位添加到最左边



对LRU很好地近似实现

页面缓冲算法(page buffering)

- 它是对FIFO算法的发展，通过被置换页面的缓冲，有机会找回刚被置换的页面；
- 被置换页面的选择和处理：用FIFO算法选择被置换页，把被置换的页面放入两个链表之一。即：如果页面未被修改，就将其归入到空闲页面链表的末尾，否则将其归入到已修改页面链表。

页面缓冲算法(page buffering)

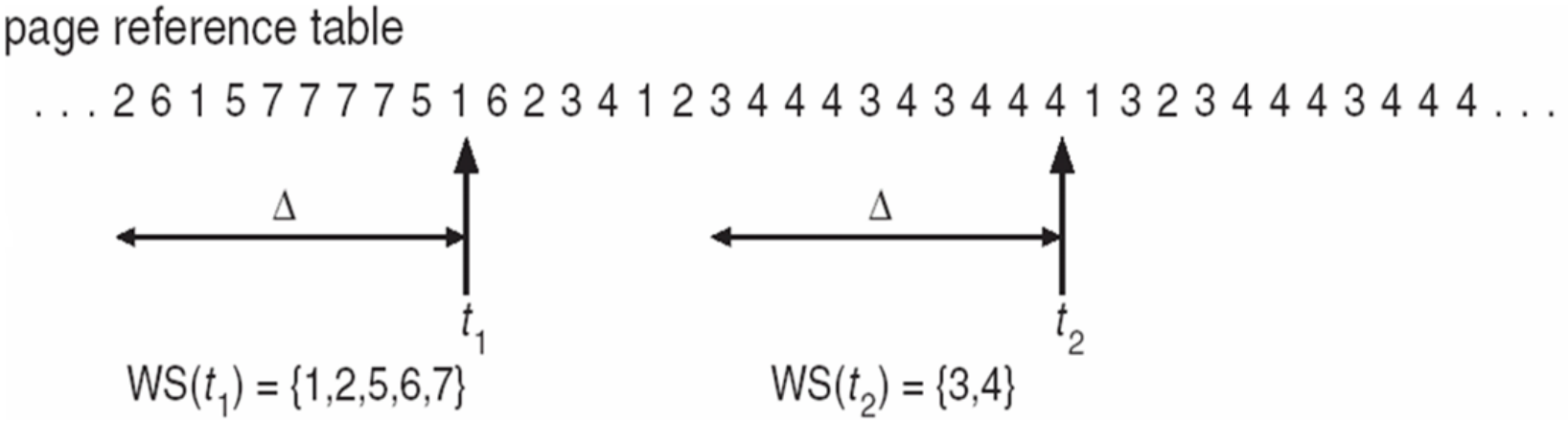
- 空闲页面和已修改页面，仍停留在内存中一段时间，如果这些页面被再次访问返回进程的内存页。
- 需要调入新的物理页面时，将新页面内容读入到空闲页面链表的第一项所指的页面，然后将第一项删除。
- 当已修改页面达到一定数目后，再将它们一起调出到外存，然后将它们归入空闲页面链表，这样能大大减少I/O操作的次数。

工作集策略(working set)

- 1968年Denning根据局部性原理提出
- 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。
- 根据时间局部性，如果能够预知程序在某段时间间隔内要访问哪些页面，并能提前将它们调入内存，将会大大降低缺页率，减少置换，提高CPU利用率
- 工作集的定义：进程在任一个时间t，都存在一个集合，包含所有最近k次内存访问过的页面，这个集合W(k,t)即称为工作集

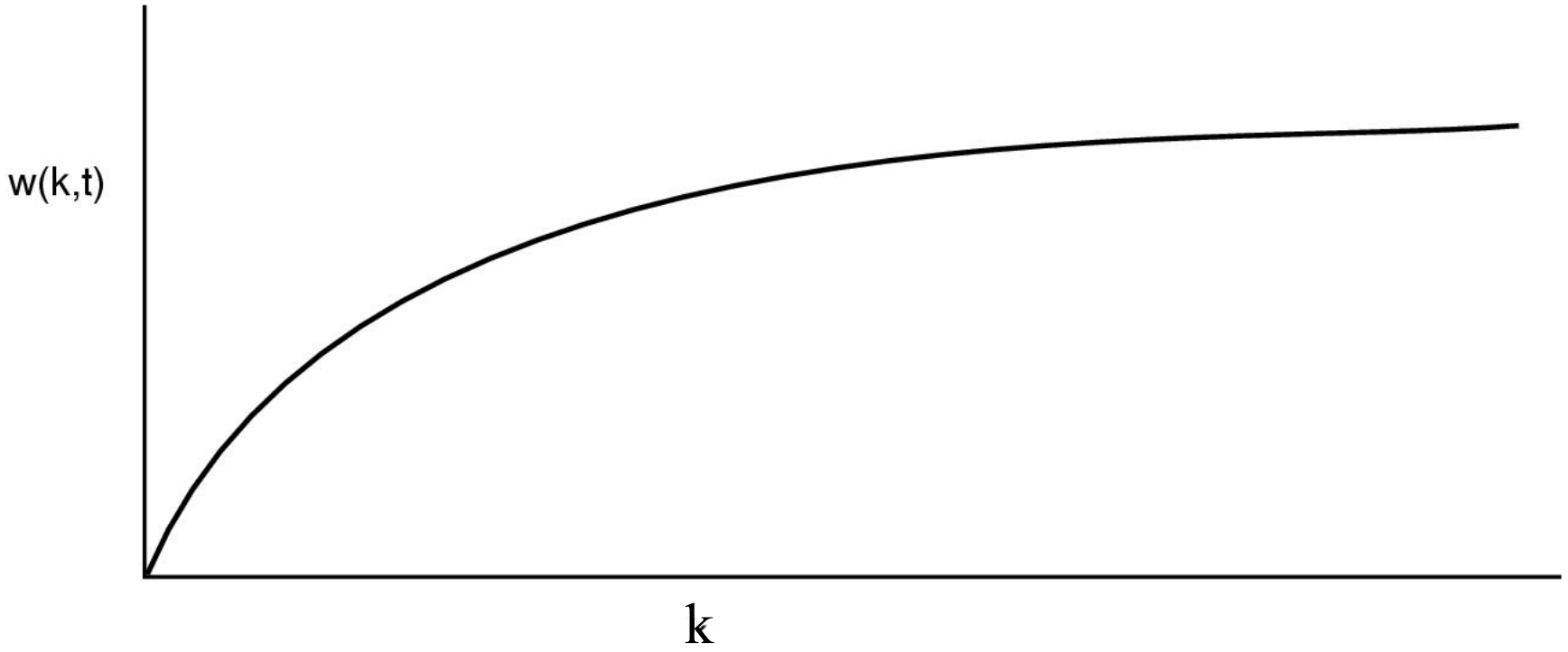
工作集策略(working set)

- 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区域的位置大致稳定时，工作集大小也大致稳定；局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值。



工作集函数随着k的变化

K在很大的范围内，工作集并不变化。



W(k,t) 是在时刻t，k次最近内存访问到的页面数

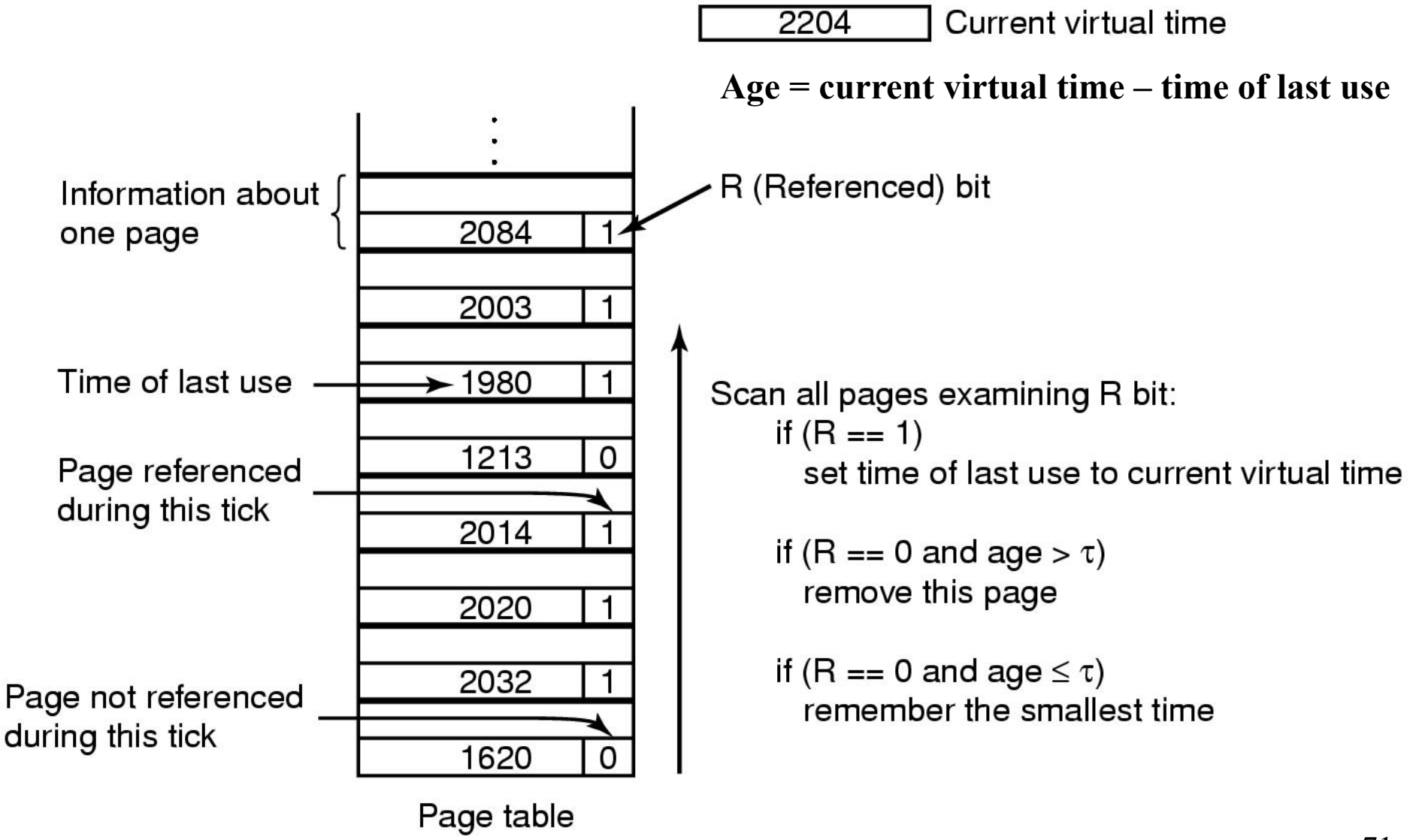
工作集策略(working set)

- 工作集算法：当发生缺页的时候，淘汰一个不在工作集的页面。
- 给定k值，如何确定工作集?

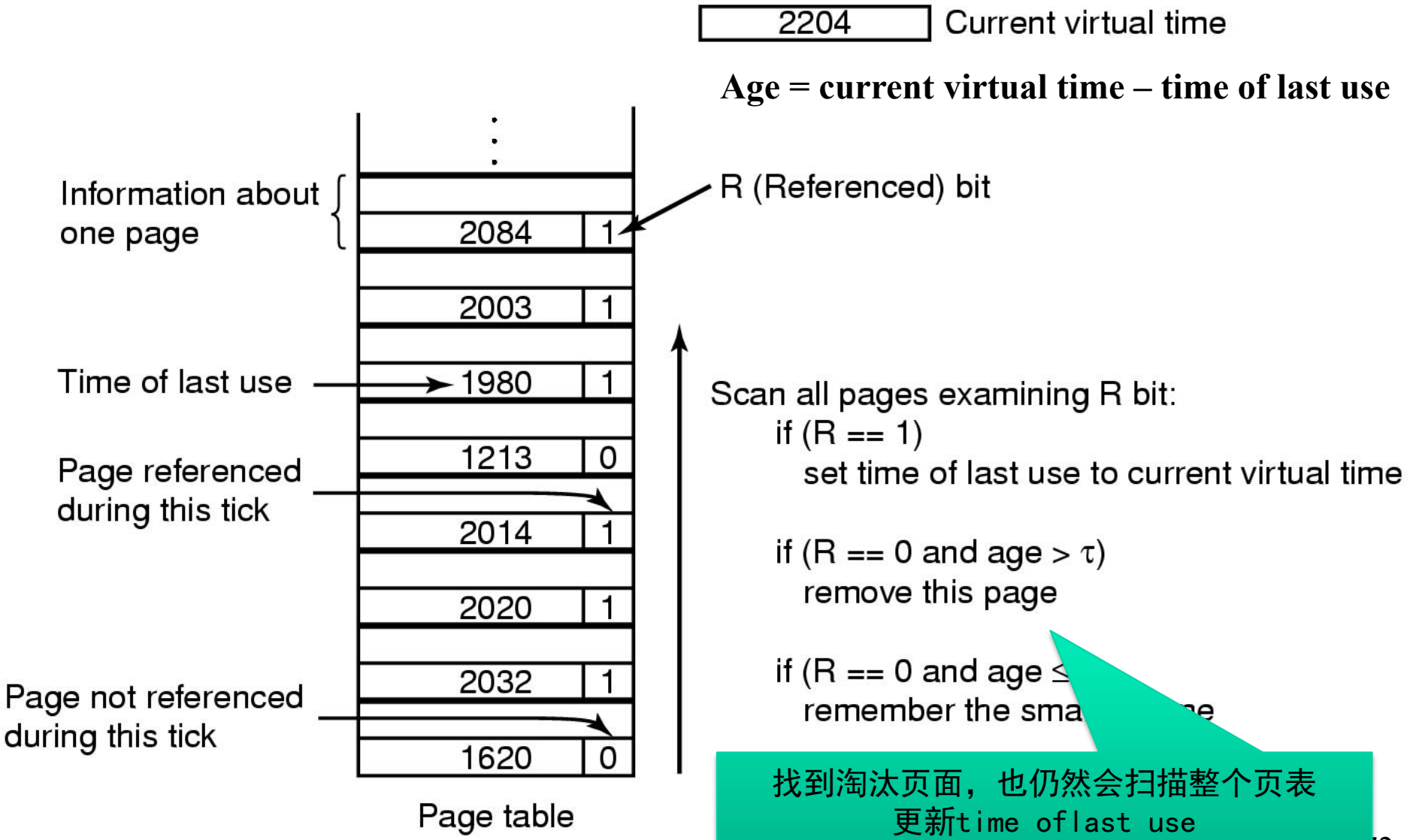
工作集策略(working set)

- 用时间间隔作为最近k次访问的近似
- 工作集的等价定义：在某段时间间隔Δ里，进程实际要访问的页面集合。
  - 可用一个二元函数W(t, Δ)表示，其中：t是执行时刻；Δ是时间窗口尺寸(window size)；工作集是在[t - Δ, t]时间段内所访问的页面的集合，| W(t, Δ) | 指工作集大小即页面数目；

基于工作集的页面置换算法



基于工作集的页面置换算法

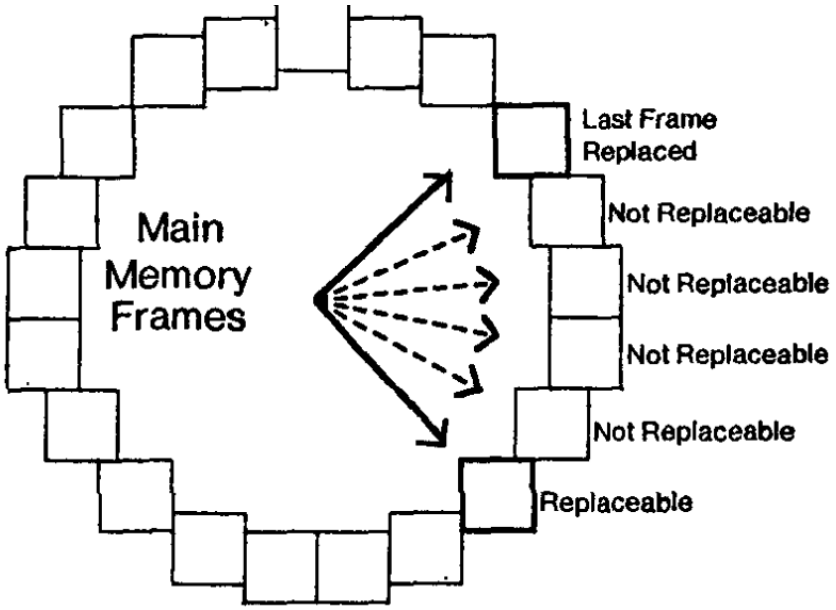


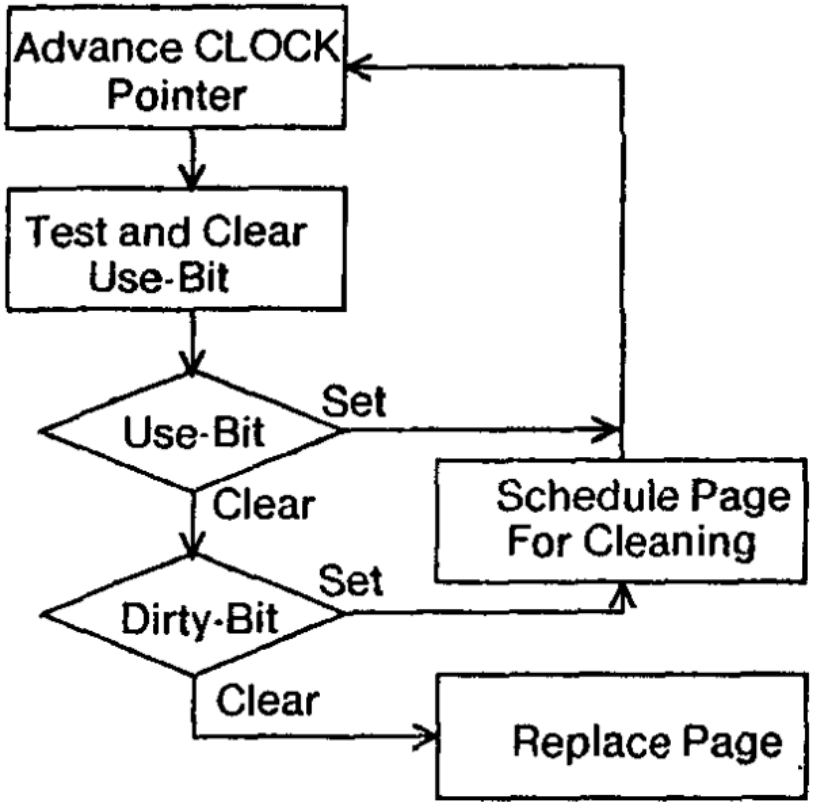
困难：

- 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；
- Δ的大小选择：
  - Δ过大，虽然进程不容易缺页，但是存储利用率低
  - Δ过小，会容易频繁产生缺页中断，降低了吞吐量

Clock算法：

- Multics上使用，全局算法，近似LRU
- 实现简单效率高





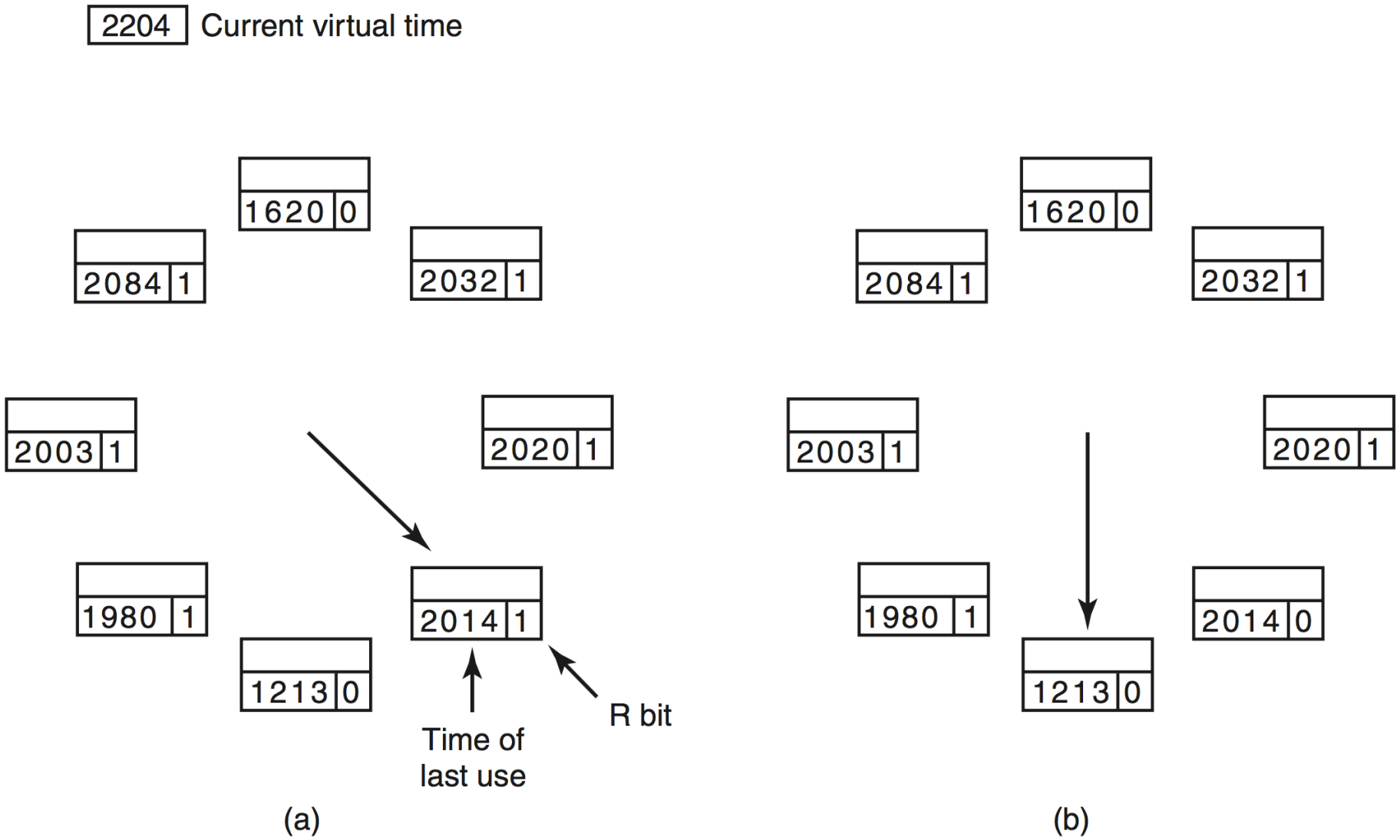
WSClock(Carr1981)：

- clock算法+工作集信息
- 实现简单，性能好
- 空间：
  - 保存页框循环表，而不是所有页表
  - 不需要保存上次访问时间
- 不需要扫描整个页表，仅仅在页框循环表中循环

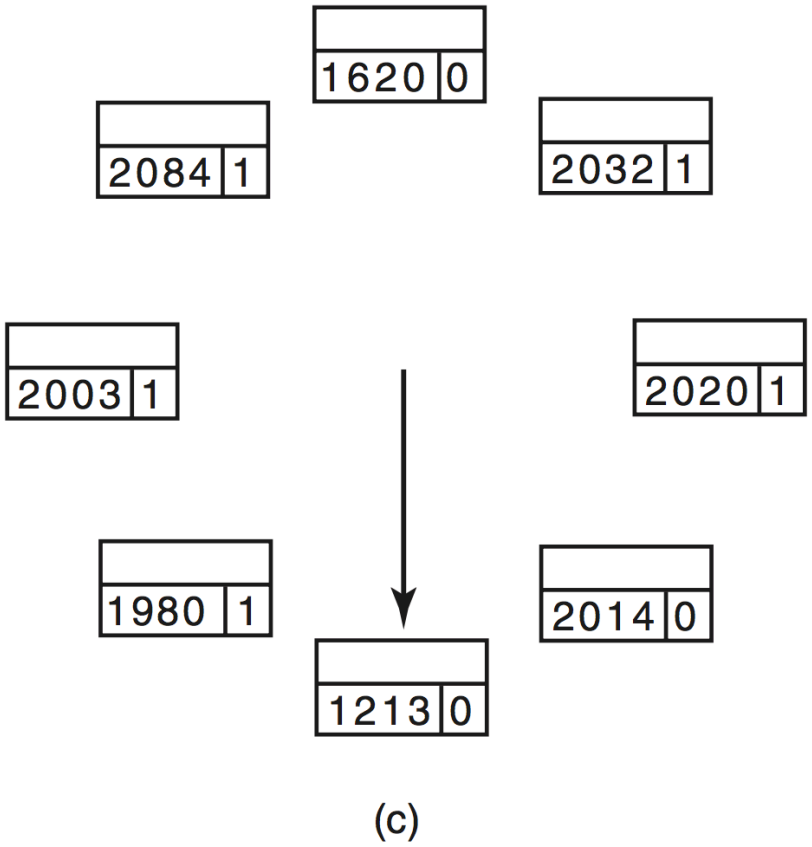
WSClock：

- 基本流程
  - 时钟中断时，检查当前指针指向页面：
  - 如果R=1，被使用过，不淘汰。置位，指针向下走，重复。
  - 如果R=0，没使用过
    - 如果age>t, 工作集外，如果m=0, 干净，不需要回写，可以直接淘汰。但是如果m=1, 不淘汰，先异步调度写磁盘，继续向前走。
  - 如果走了一圈了。。。
    - 如果发出过调度指令，那么继续走，会遇到写完的干净的页面
    - 如果没有调度过，随机替换一个干净页面。无干净页面，替换当前页。

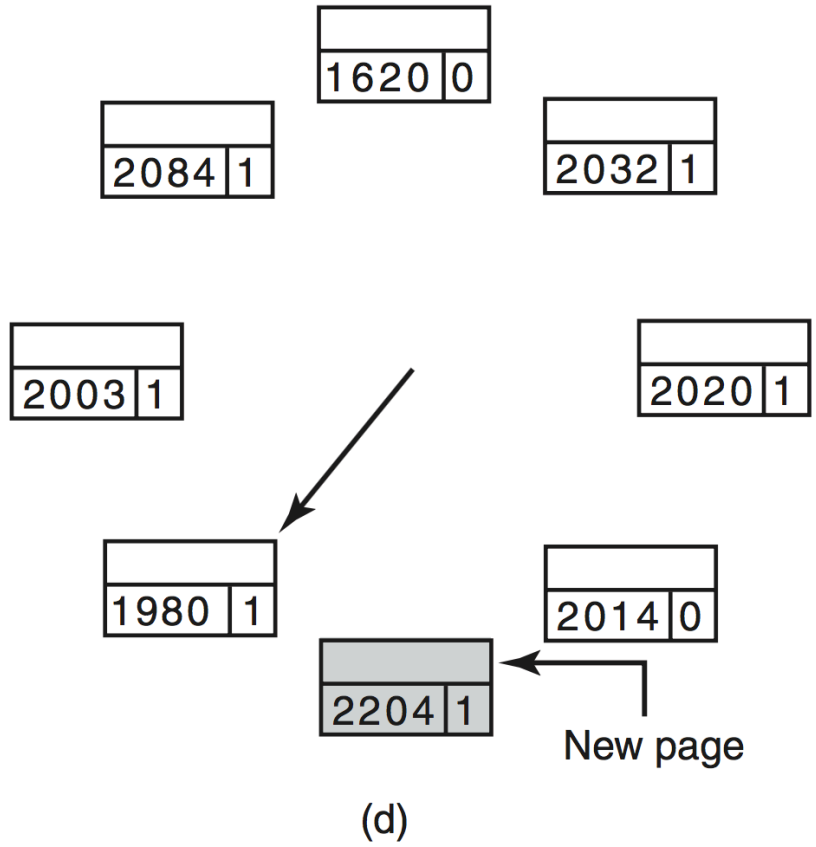
WSClock：


2204
Current virtual time
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 1
1213 0
Time of last use
R bit
(a)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
1213 0
(b)

WSClock：


1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
1213 0

(c)


1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
2204 1
New page

(d)

WSClock

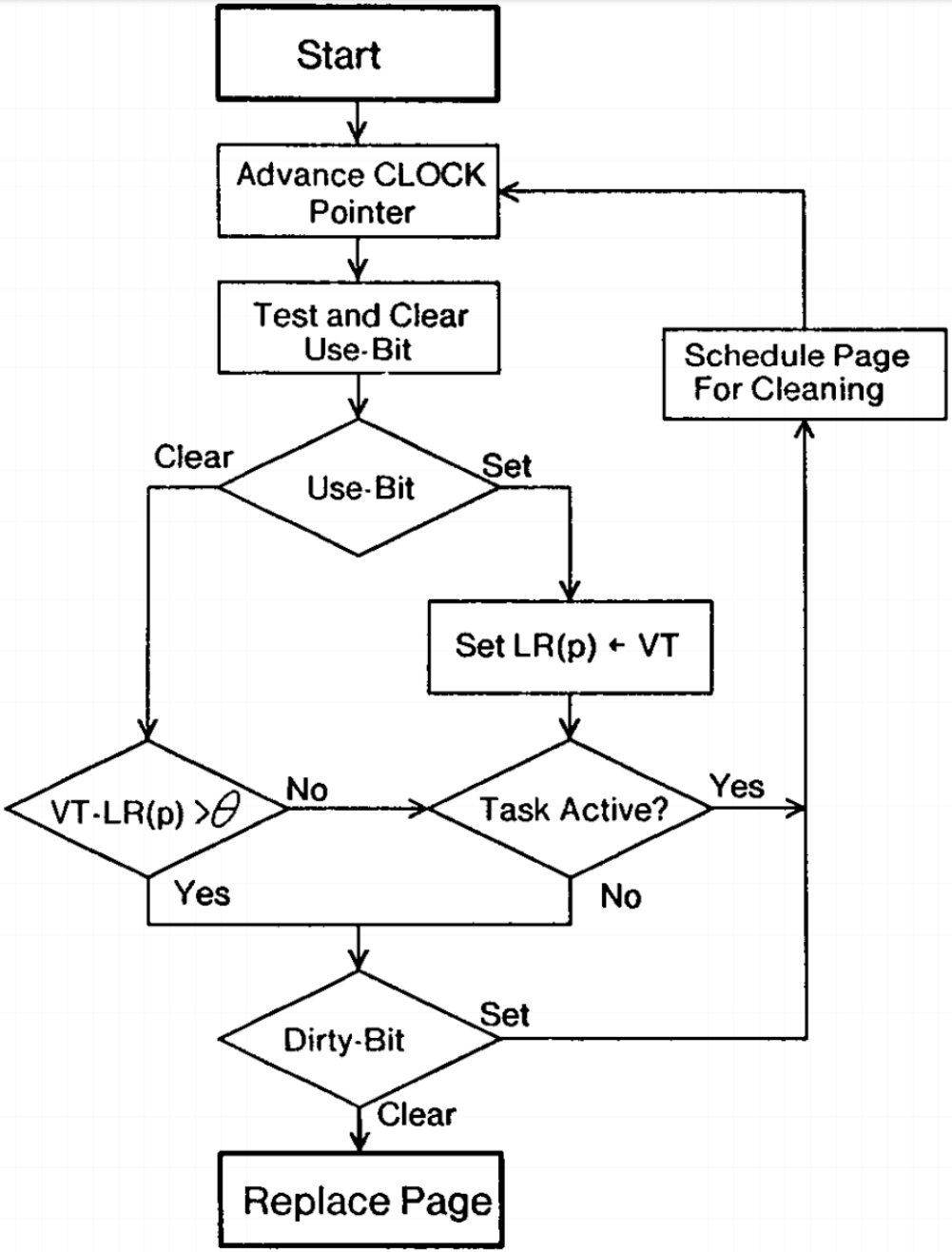


Figure 2. WSCLOCK Replacement Algorithm

典型算法的比较

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**

缺页中断率

- 假定一个程序共有n页，系统分配给它的内存块是m块（m、n均为正整数，且1≤ m ≤n）。因此，该程序最多有m页可同时被装入内存。如果程序执行中访问页面的总次数为A，其中有F次访问的页面尚未装入内存，故产生了F次缺页中断。现定义：

$$f = F / A$$

把f称为“缺页中断率”。

影响缺页中断率的因素

- 分配给程序的内存块数
  - 分配给程序的内存块数多，则同时装入内存的页面数就多，故减少了缺页中断的次数，也就降低了缺页中断率。反之，缺页中断率就高。
- 页面的大小
  - 页面的大小取决于内存分块的大小，块大则页面也大，每个页面大了则程序的页面数就少。装入程序时是按页存放在内存中的，因此，装入一页的信息量就大，就减少了缺页中断的次数，降低了缺页中断率。反之，若页面小则缺页中断率就高。

影响缺页中断率的因素

- 程序编制方法
  - 怎样编制程序也是值得探讨的，程序编制的方法不同，对缺页中断的次数有很大影响。
  - 例如，有一个程序要把128×128的数组置初值“0”，数组中的每个元素为一个字。现假定页面的大小为每页128个字，数组中的每一行元素存放在一页中。能供这个程序使用的内存块只有一块，开始时把第一页装入了内存。若程序如下编制：

```
VAR A：ARRAY [1··128，1··128] OF Integer;
FOR j:= 1 TO 128 DO
    FOR i:= 1 TO 128 DO
        A[i, j]:= 0;
```

影响缺页中断率的因素

```
VAR A：ARRAY [1··128，1··128] OF Integer；

FOR i:= 1 TO 128 DO

FOR j:= 1 TO 128 DO

A[i, j]:= 0；
```

- 每装入一页后就对一行元素全部清“0”后才产生缺页中断，故总共产生（128—1）次缺页中断。
- 缺页中断率与程序的局部化程度密切相关。
- 一般来说，希望编制的程序能经常集中在几个页面上进行访问，以减少缺页中断率。

全局分配策略和缺页率

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

初始页面配置

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

局部分配策略

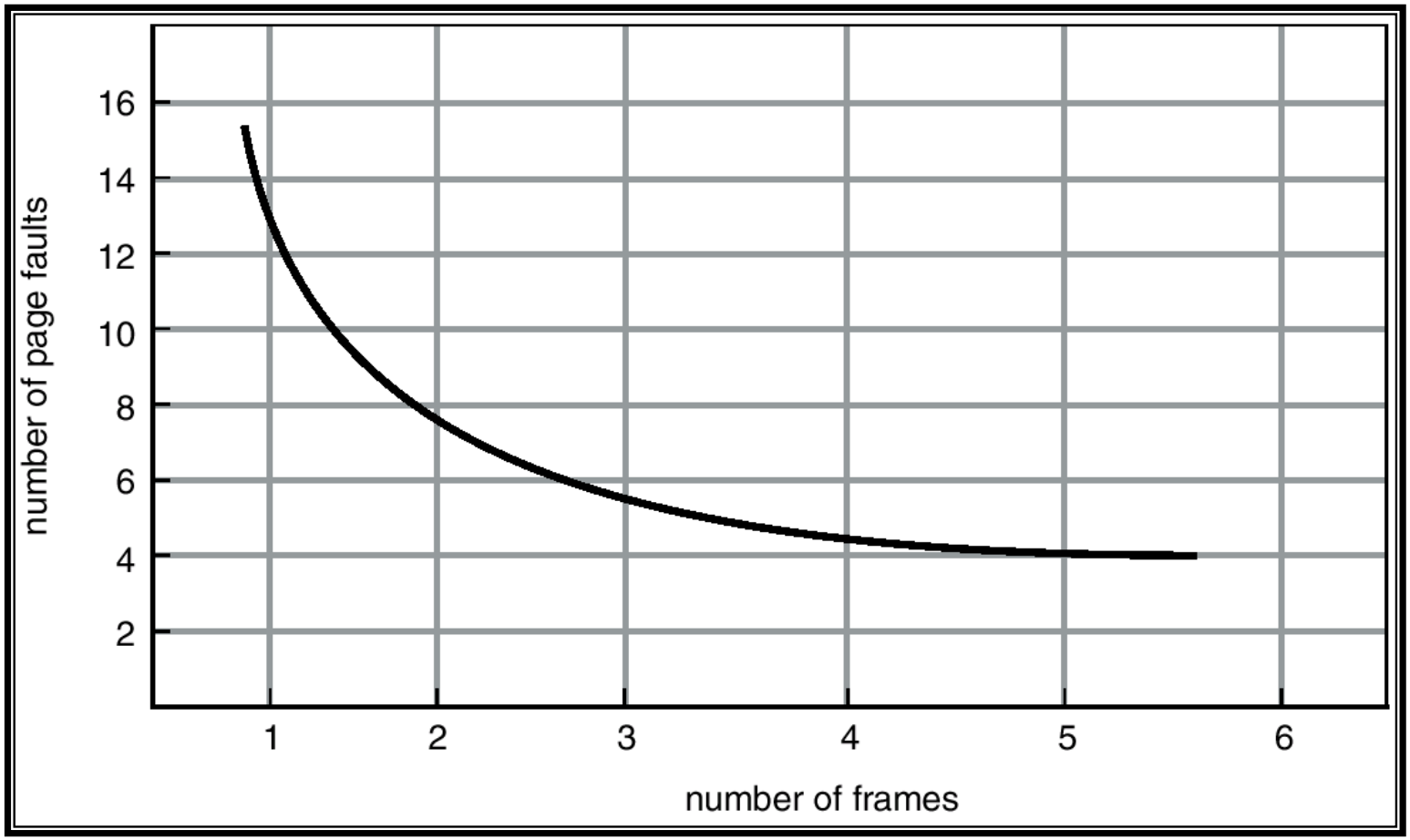
A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

全局分配策略

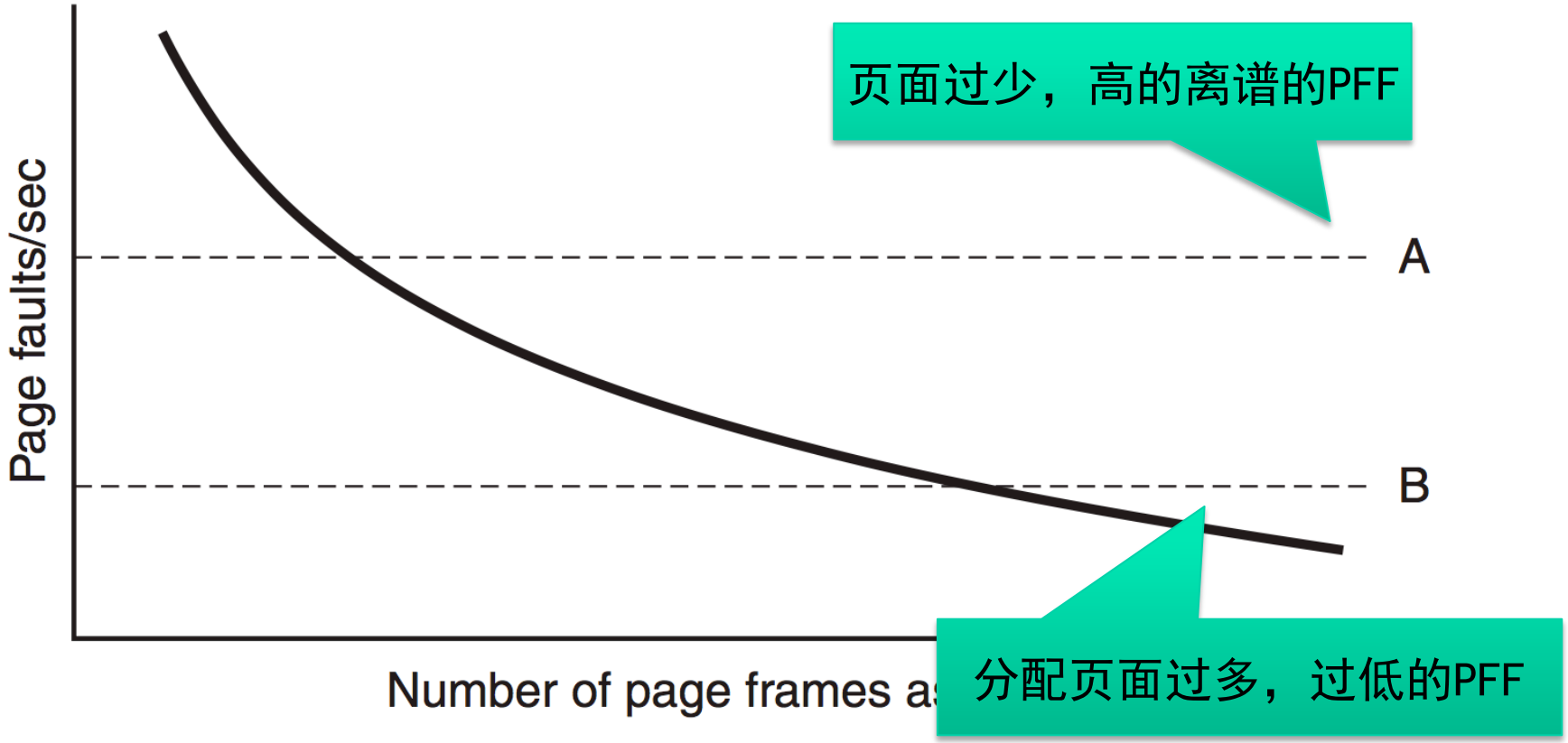
缺页中断率（PFF）算法的假设

- 理想的情况：缺页率随页框数增加而下降



使用缺页中断率算法确定内存分配

- 什么时候增加或者减少分配给进程的页面?



置换算法与分配策略的关系

- 同时适合局部分配策略和全局分配策略
  - FIFO： 替换所有内存（v.s. 当前进程）最老的页面
  - LRU：将所有内存（v.s. 当前进程）最近最少页面替换
- 仅仅适合局部分配
  - 工作集和WSClock
  - 工作集其实是和特定进程对应的
  - 并不存在整个机器的工作集

抖动问题(thrashing)

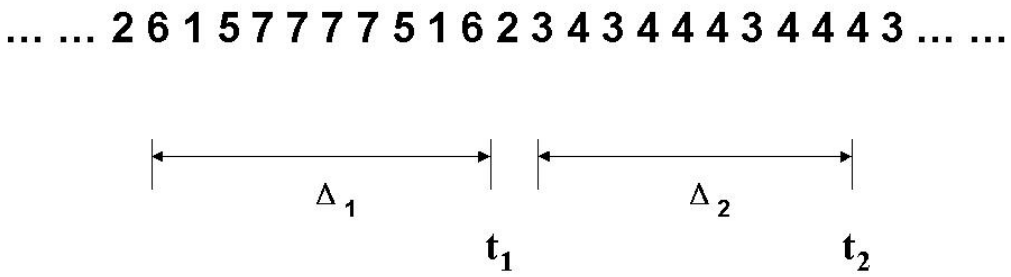
- 页面调度算法对缺页中断率的影响也很大，调度不好就会出现“抖动”。
- 随着驻留内存的进程数目增加，或者说进程并发水平(multiprogramming level)的上升，处理器利用率先是上升，然后下降。
- 这里处理器利用率下降的原因通常称为虚拟存储器发生“抖动”，也就是：每个进程的常驻集不断减小，缺页率不断上升，频繁调页使得调页开销增大。
- OS要选择一个适当的进程数目，以在并发水平和缺页率之间达到一个平衡。

抖动的预防

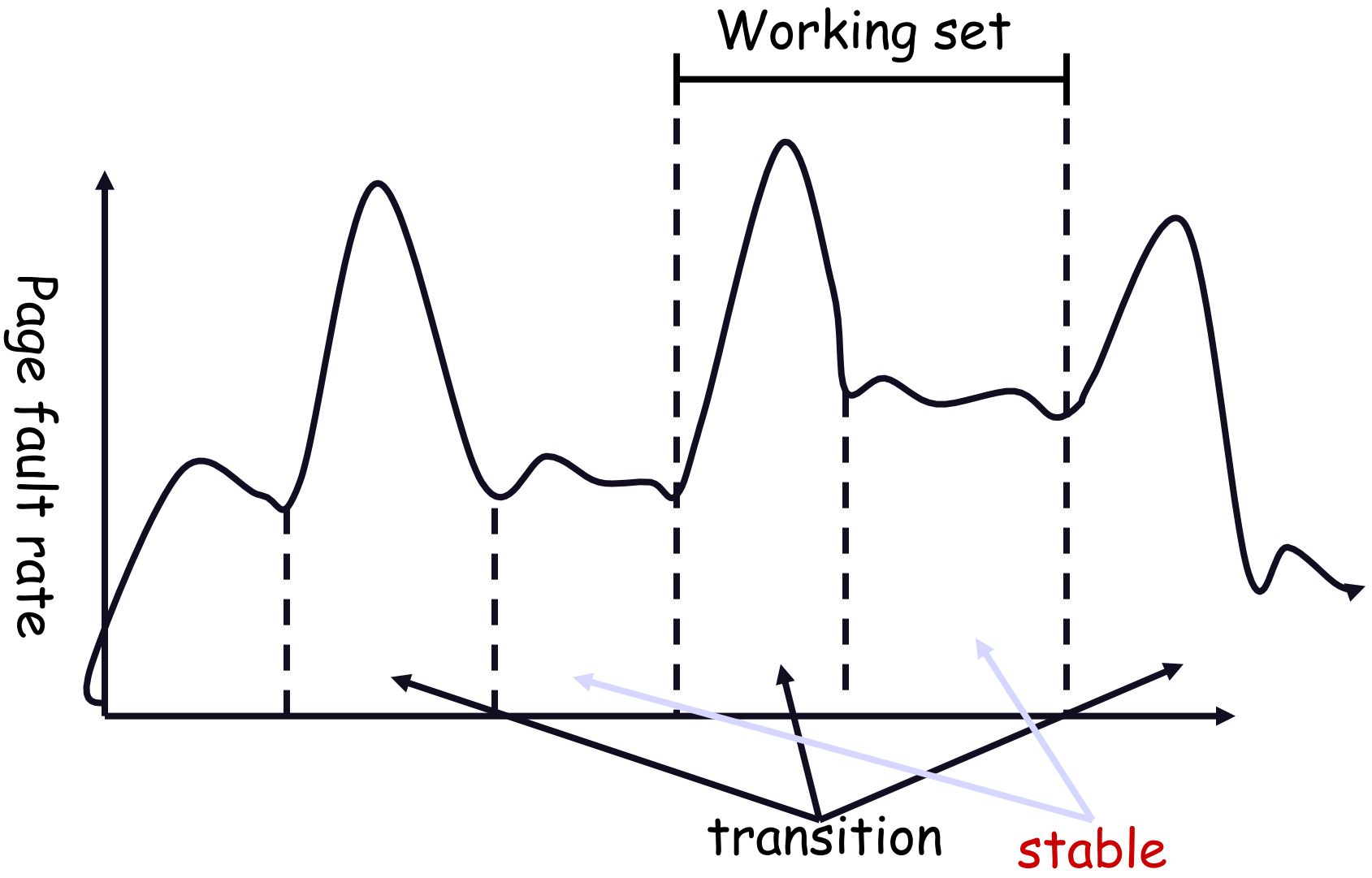
- 局部置换策略
- 引入工作集算法
- 预留部分页面
- 挂起若干进程

抖动的预防-工作集

- 由模拟实验知道，任何程序对内存都有一个临界值要求，当分配给进程的物理页面数小于这个临界值时，缺页率上升；当分配给进程的物理页面数大于该临界值时，再增加物理页面数也不能显著减少缺页次数。因此，希望分配给进程的物理页面数与当前工作集大小一致。
- 在实现时，操作系统为每一个进程保持一个工作集，并为该进程提供与工作集大小相等的物理页面数，这一过程可动态调整。统计工作集大小一般由硬件完成，系统开销较大。



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。

置换算法的例子

例1 （a） 某程序在内存中分配三个页面，初始为空，所需页面的走向为4，3，2，1，4，3，5，4，3，2，1，5，采用FIFO，LRU，OPT算法，请计算整个缺页次数。

置换算法的例子（FIFO）

在下面的图中，“时间长-页”表示在内存时间最长的页面，“时间中-页”其次，“时间短-页”表示未使用时间最短的页面。X表示缺页，√表示不缺页。。

页面走向	4	3	2	1	4	3	5	4	3	2	1	5
时间短-页	4	3	2	1	4	3	5	5	5	2	1	1
时间中-页		4	3	2	1	4	3	3	3	5	2	2
时间长-页			4	3	2	1	4	4	4	3	5	5
	x	x	x	x	x	x	x	√	√	x	x	√
	1	2	3	4	5	6	7			8	9	

置换算法的例子 (LRU)

在上述例子中采用LRU算法，请计算整个缺页次数。

在下面的图中，“时间长-页”表示在未使用时间最长的页面，“时间中-页”其次，“时间短-页”表示在内存时间最短的页面。X表示缺页，√表示不缺页。

页面走向	4	3	2	1	4	3	5	4	3	2	1	5
时间短-页	4	3	2	1	4	3	5	4	3	2	1	5
时间中-页		4	3	2	1	4	3	5	4	3	2	1
时间长-页			4	3	2	1	4	3	5	4	3	2
	x	x	x	x	x	x	x	√	√	x	x	x
	1	2	3	4	5	6	7			8	9	10

北京航 共缺页中断 10 次。

置换算法的例子 (OPT)

- 在上述例子中采用OPT 算法，请计算整个缺页次数。
- 最长时间以后才会用到的页面，用“时间长-页”表示，“时间中-页”表示其次，“时间短-页”表示最短时间会用到的页面。

页面走向	4	3	2	1	4	3	5	4	3	2	1	5
时间短-页	4	3	2	1	1	1	5	5	5	2	1	1
时间中-页		4	3	3	3	3	3	3	3	5	5	5
时间长-页			4	4	4	4	4	4	4	4	4	4
	x	x	x	x	√	√	x	√	√	x	x	√
	1	2	3	4			5			6	7	

北京航空 共缺页中断 7 次。

缺页中断的计算

- 例2 某程序在内存中分配m页初始为空，页面走向为1，2，3，4，1，2，5，1，2，3，4，5。当m＝3，m＝4时，试用FIFO算法计算缺页中断分别为多少次？说明所出现的现象。

缺页中断的计算

- 例2 某程序在内存中分配m页初始为空，页面走向为1，2，3，4，1，2，5，1，2，3，4，5。当m＝3，m＝4时，试用FIFO算法计算缺页中断分别为多少次？说明所出现的现象。
  - 通过计算得到当m＝3时，缺页中断9次；m＝4时，缺页中断10次。

123412512345

****　　*　****

123412512345

****　　******

  - 我们发现，当分配给进程的物理页面数增加时，缺页次数反而增加。这一现象称为Belady异常现象，FIFO页面置换算法会产生异常现象。

问题

- 应用是否可以修改自己的页表?

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