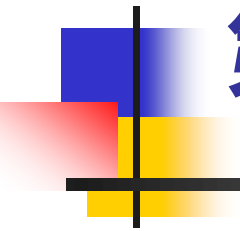
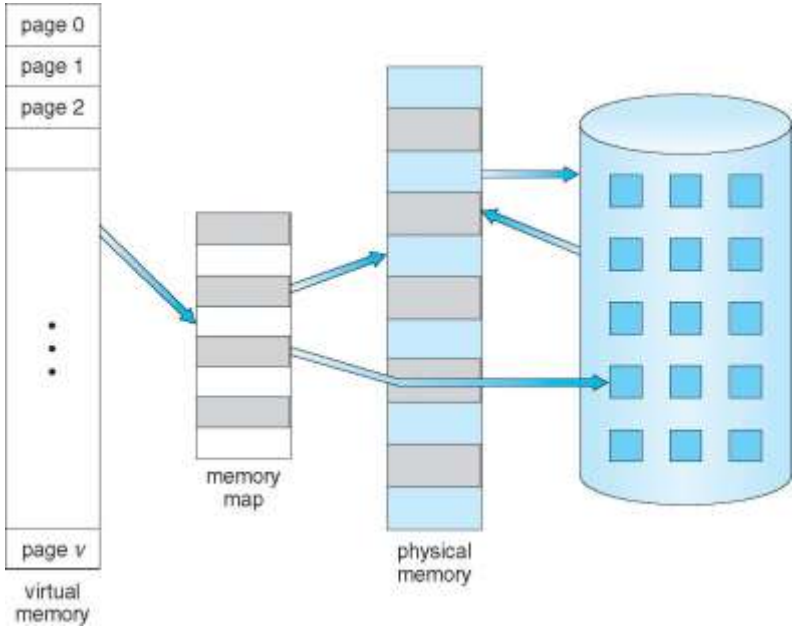


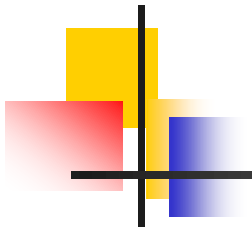
第7章 虚拟内存管理

- 前章中的内存管理方法
  - 物理存储器管理方式要求作业运行前全部装入内存
  - 作业装入内存后一直驻留内存直至运行结束，或者该作业被换出。
- 缺点：
  - 这种物理存储器管理方式限制了大程序的运行。
  - 虚拟地址空间也远远大于物理地址空间



关于程序执行的特点

1. 程序中只有少量分支和过程调用，大都是顺序执行的
   - 即，要取的下一条指令紧跟在当前执行指令之后。
2. 程序往往包含若干个循环
   - 这些是由相对较少的几个指令重复若干次组成的，在循环过程中，计算被限制在程序中一个很小的相邻部分中（如计数循环）。
3. 很少会出现连续不断的过程调用序列
   - 相反，程序中过程调用的深度限制在一个小的范围内，因而，一段时间内，指令引用被局限在很少几个过程中。
4. 对于连续访问数组之类的数据结构操作
   - 往往是对存储区域中的数据或相邻位置的数据（如动态数组）的操作。
5. 程序中有些部分是彼此互斥的，不是每次运行时都用到的
   - 如出错处理程序，仅当在数据和计算中出现错误时才会用到，正常情况下，出错处理程序不放在主存，不影响整个程序的运行。



第7章 虚拟内存管理

7.1 虚拟内存概念

虚拟内存概念

- 上述种种情况说明，作业执行时没有必要把全部信息同时存放在主存储器中，而仅仅只需装入一部分。这一原理称为“局部性原理”。

- 局部性原理
  - CPU访问存储器时，无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的连续区域中。

局部性的体现

- 局部性体现为：
  - 时间局部性：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间内。
  - 空间局部性：当前执行的指令和将要执行的指令，当前访问的数据和将要访问的数据，都集中在一个较小范围内。
  - 顺序局部性：顺序执行与跳转比例5：1

- 虚拟内存的理论基础
  - 程序执行时的局部性原理。

虚拟内存的基本原理

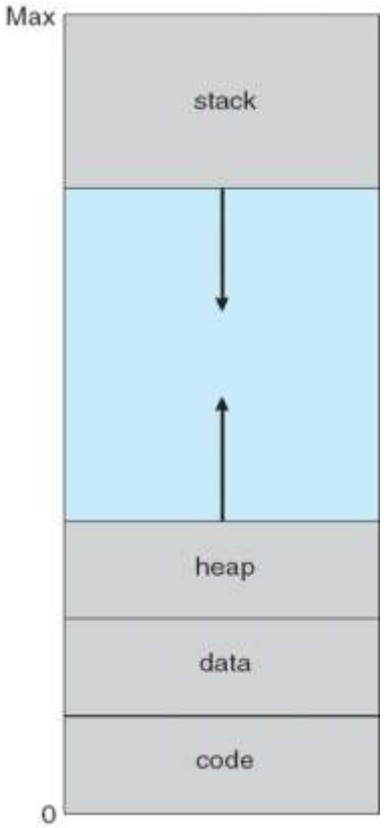
- 运行前部分装入
  - 在程序运行之前，将程序的一部分放入内存后就启动程序执行。
- 运行时的换入与换出
  - 在程序执行过程中，当所访问的信息不在内存时，由OS将所需要的部分调入内存，然后继续执行程序。
  - OS将内存中暂时不使用的内容换出到外存上，从而腾出空间存放将要调入内存的信息。
- 从效果上看，这样的计算机系统好像为用户提供了一个存储容量比实际内存大得多的存储器，将这个存储器称为虚拟内存。

虚拟内存的定义

- 虚拟内存，也称为虚拟存储器
  - 指具有请求调入和替换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。
- 虚拟内存是一种以时间换空间的技术。
- 特征
  - 离散性：不连续内存分配
  - 多次性：一个作业分多次装入内存
  - 对换性：允许运行中换进换出
  - 虚拟性：逻辑上扩充内存

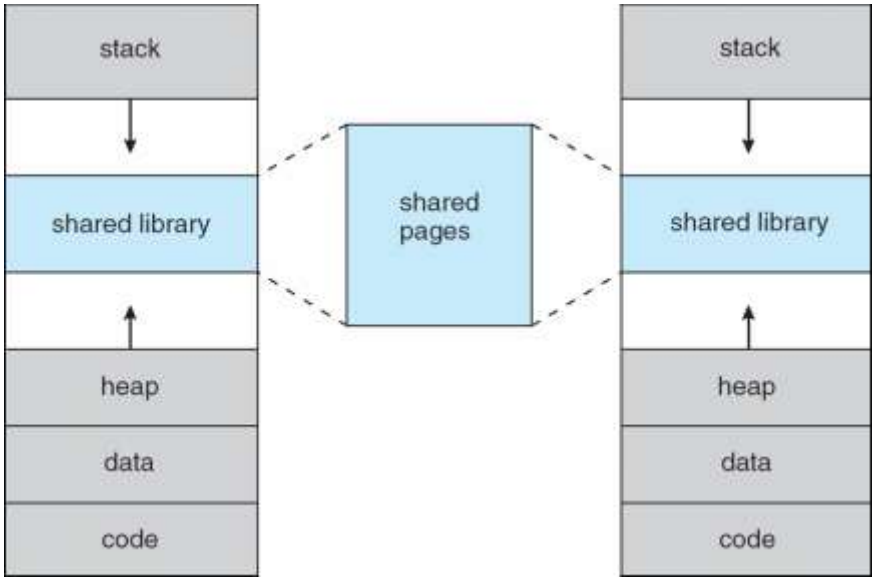
虚拟内存的优势

- 节约空间
  - Stack和heap的虚拟地址之间有巨大的空余空间，但无需立即分配



虚拟内存的优势

- 虚拟内存可为多个进程实现文件、内存的共享
  - 系统库可以通过虚拟地址空间映射，实现多个进程的共享，**只读方式**
  - 在fork时，通过共享页面，可以加速进程的创建
  - 为多个进程提供内存共享，如实现进程通信



虚拟内存的本质

- 虚拟内存的本质
  - 将程序的访问地址和内存的可用地址分离，为用户提供一个大于实际主存的虚拟内存。

- 虚拟内存的容量受限于：
  - 地址结构
  - 外存容量

实现虚拟存储技术的物质基础

- 相当数量的外存：
  - 足以存放多个用户的程序。
- 一定容量的内存：
  - 在处理机上运行的程序必须有一部分信息存放在内存中。
- 地址变换机构：
  - 动态实现逻辑地址到物理地址的变换。

实现虚拟存储技术的挑战

- 主存和辅存统一管理问题
- 逻辑地址和物理地址的转换问题
- 部分装入和部分对换的问题

虚拟内存的实现方法

- 常用的虚拟存储技术有：
  - 请求分页存储管理
  - 请求分段存储管理

虚拟内存的实现方法

- 常用的虚拟存储技术有：
  - 请求分页存储管理
  - 请求分段存储管理

- 前章与本章的区别
  - 非请求分页/段：一次性调入
  - 请求分页/段：按需调入，不需的换出

7.2 请求分页存储管理

- 请求分页存储管理方法是在分页存储管理的基础上增加了请求调页和页面替换功能。

- 实现思想：
  - 部分装入：在作业运行之前只装入当前需要的一部分页面便启动作业运行。
  - 按需装入：在作业运行过程中，若发现所要访问的页面不在内存，便由硬件产生缺页中断，请求OS将缺页调入内存。
  - 页面替换/页面置换：若内存无空闲存储空间，则根据某种替换算法淘汰已在内存的某个页面，以腾出内存空间装入缺页。

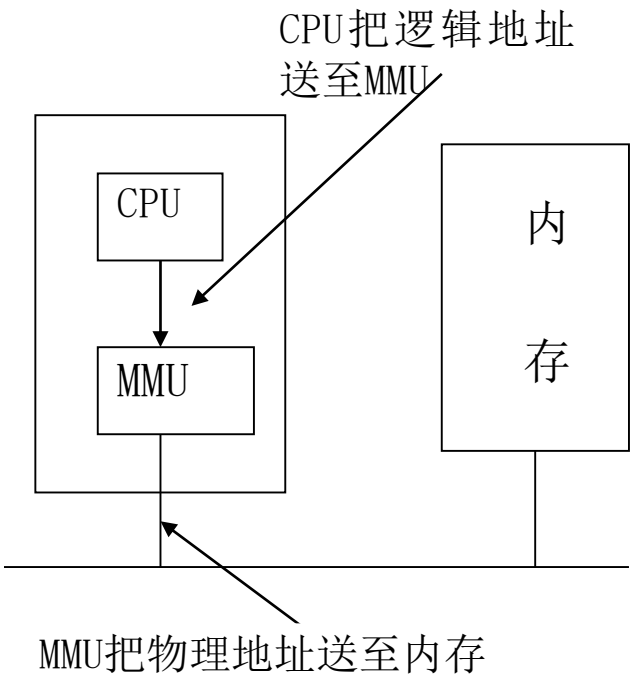
请求分页系统的支持机构

- 请求分页的支持机构有：
  - 物理部件：MMU(Memory Management Unit)
  - 页表
  - 缺页中断机构
  - 地址变换机构
  - 请求调页和页面替换软件

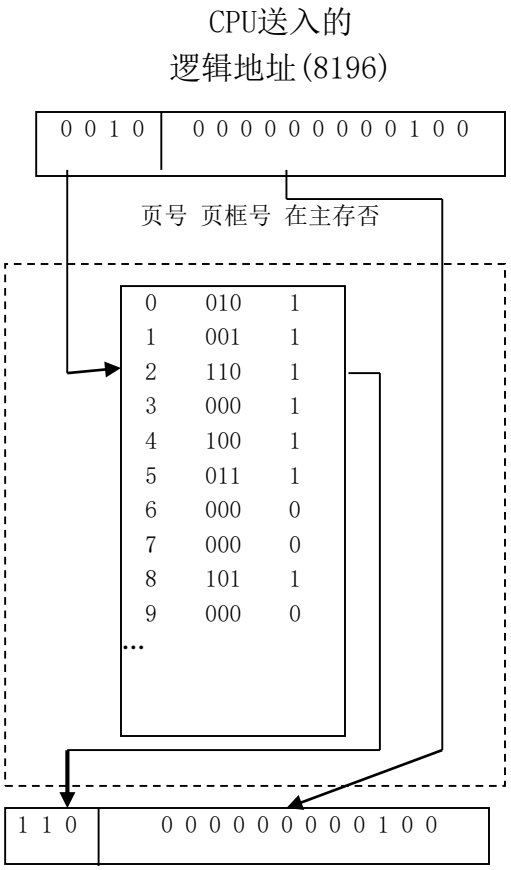
7.2.1内存管理单元MMU

- 主存管理单元MMU完成逻辑地址到物理地址的转换功能，它接受**虚拟地址作为输入，物理地址作为输出，直接送到总线上，对内存单元进行寻址**。

MMU原理



MMU的位置、功能和16个4KB页面情况下MMU的内部操作



MMU主要功能

① 管理硬件页表基址寄存器。
② 分解逻辑地址。
③ 管理快表TLB。
④ 访问页表。
⑤ 发出缺页中断或越界中断，并将控制权交给内核存储管理处理。
⑥ 管理特征位，设置和检查页表中各个特征位。

7.2.2 页表

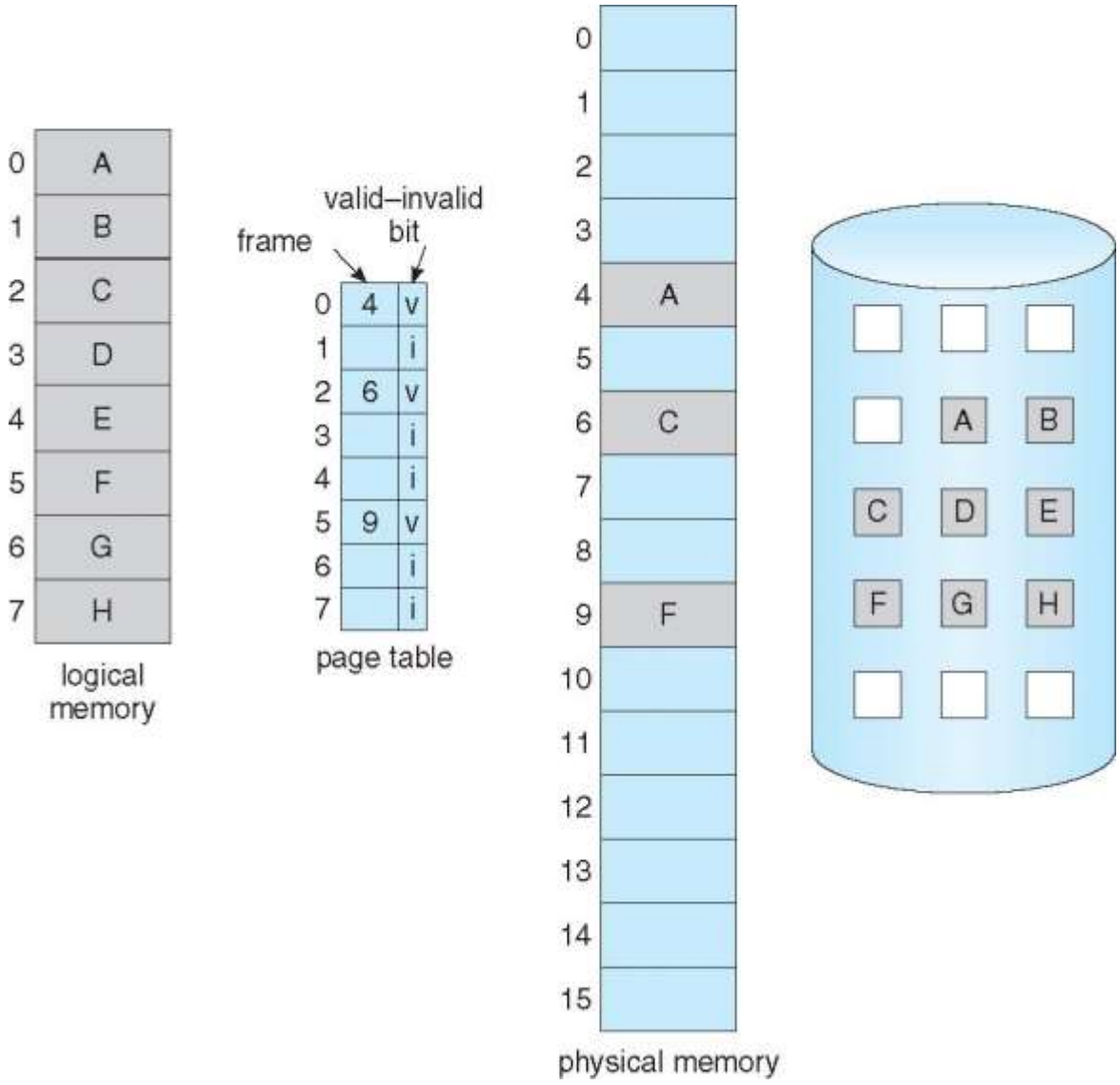
- 请求分页系统中使用的主要数据结构仍然是页表。
- 但由于每次只将作业的一部分调入内存，还有一部分内容存放在磁盘上，故需要在页表中增加若干项。
- 扩充后的页表项如下所示：

扩充后的页表项

页号	物理块号	存在位	访问字段	修改位	外存地址

- 页号和物理块号：其定义同分页存储管理。
- 存在位：用于表示该页是否在主存中。
- 访问字段：用于记录本页在一段时间内被访问的次数，或最近已有多长时间未被访问。
- 修改位：用于表示该页调入内存后是否被修改过。
- 外存地址：用于指出该页在外存上的地址。（**注：这一项往往是通过在缺页处理程序中，额外计算而获得，不需存储**）

请求分页示例图


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 G H

7.2.3 缺页中断与地址变换

- 在请求分页系统中，当所访问的页不在内存时，便产生缺页中断，请求OS将缺页调入内存

- 缺页中断处理程序根据该页在外存的地址把它调入内存
  - 在调页过程中，若内存有空闲空间，则缺页中断处理程序只需把缺页装入并修改页表中的相应项
  - 若内存中无空闲物理块，则需要先淘汰内存中的某些页
  - 若淘汰页曾被修改过，则还要将其写回外存

缺页中断的处理过程

① 发起对地址的访问，MMU到页表中检查引用情况

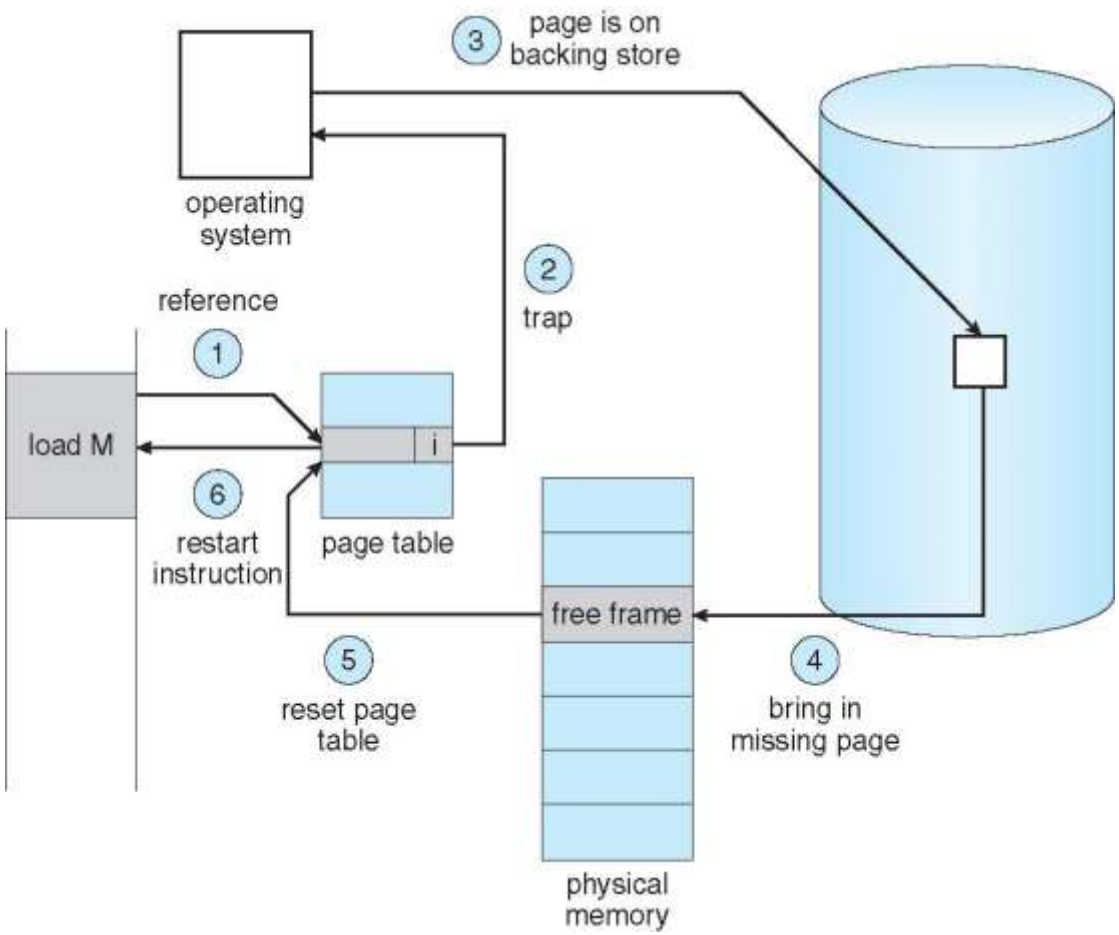
② 若不在内存，产生缺页中断，陷入缺页中断程序

③ OS在外存中寻找外存中的页面备份

④ 寻找空闲页帧，或依据某种替换算法选择被替换的页帧，将页面调入内存

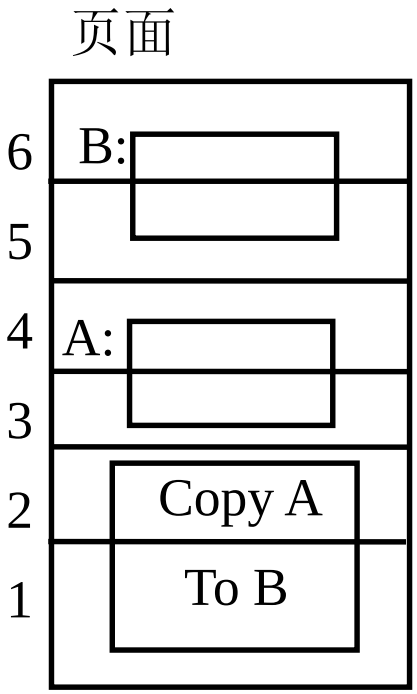
⑤ 修改页表项信息

⑥ 重新执行产生缺页的指令



缺页中断与一般中断的区别

- 缺页中断与一般中断的区别主要有：
  - 在指令的执行期间产生和处理缺页中断。
  - 一条指令可以产生多个缺页中断。如：执行一条复制指令copy A to B
  - 缺页中断返回后执行产生中断的指令，一般中断返回时执行下条指令
    - **一个特例**：Copy A to B 发生缺页中断，如果A B所包括页面地址有叠加，则不能简单返回到中断产生的指令
      - 方法1: 先访问A，B，提前触发缺页
      - 方法2: 缓存并回滚结果

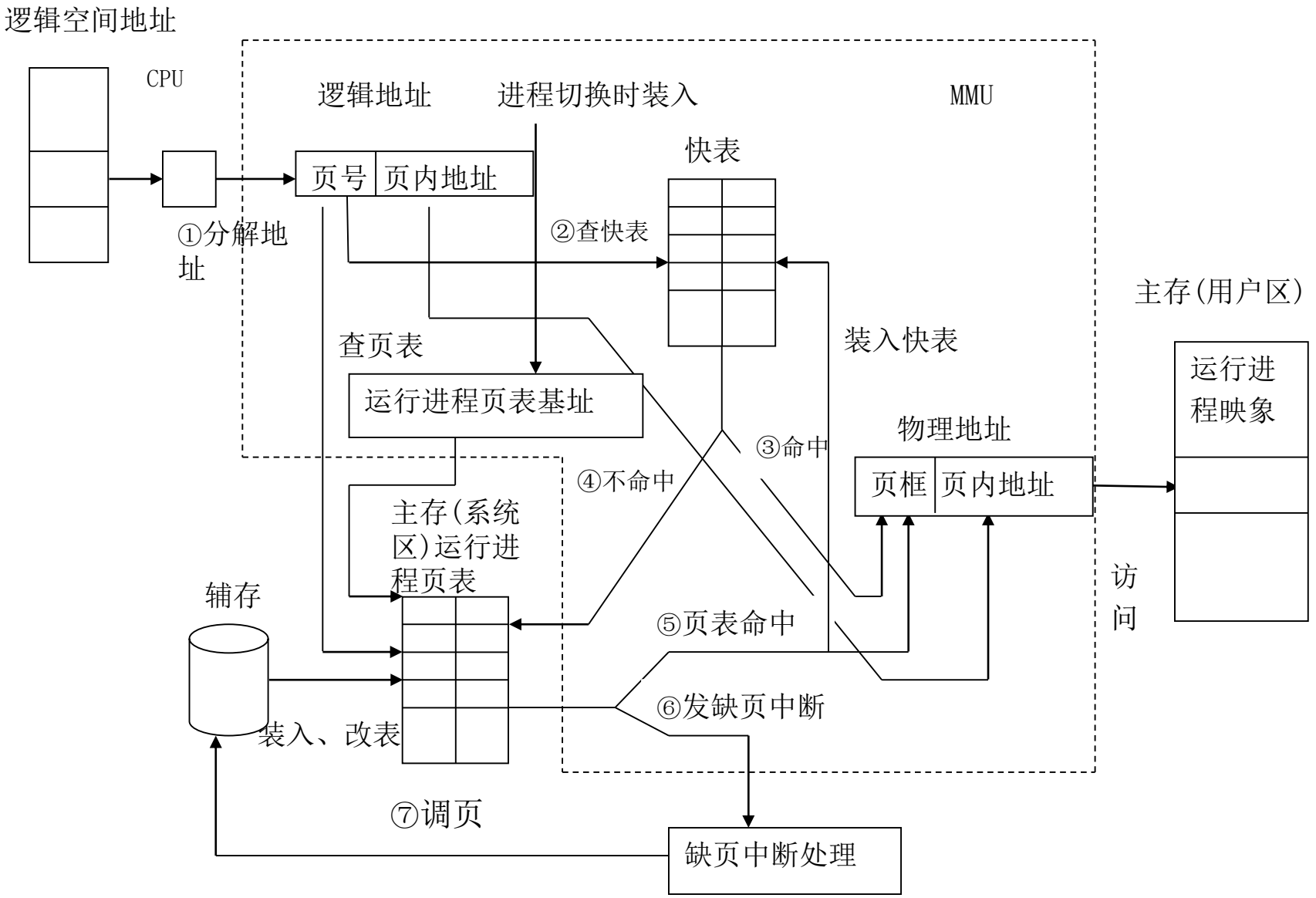


地址变换

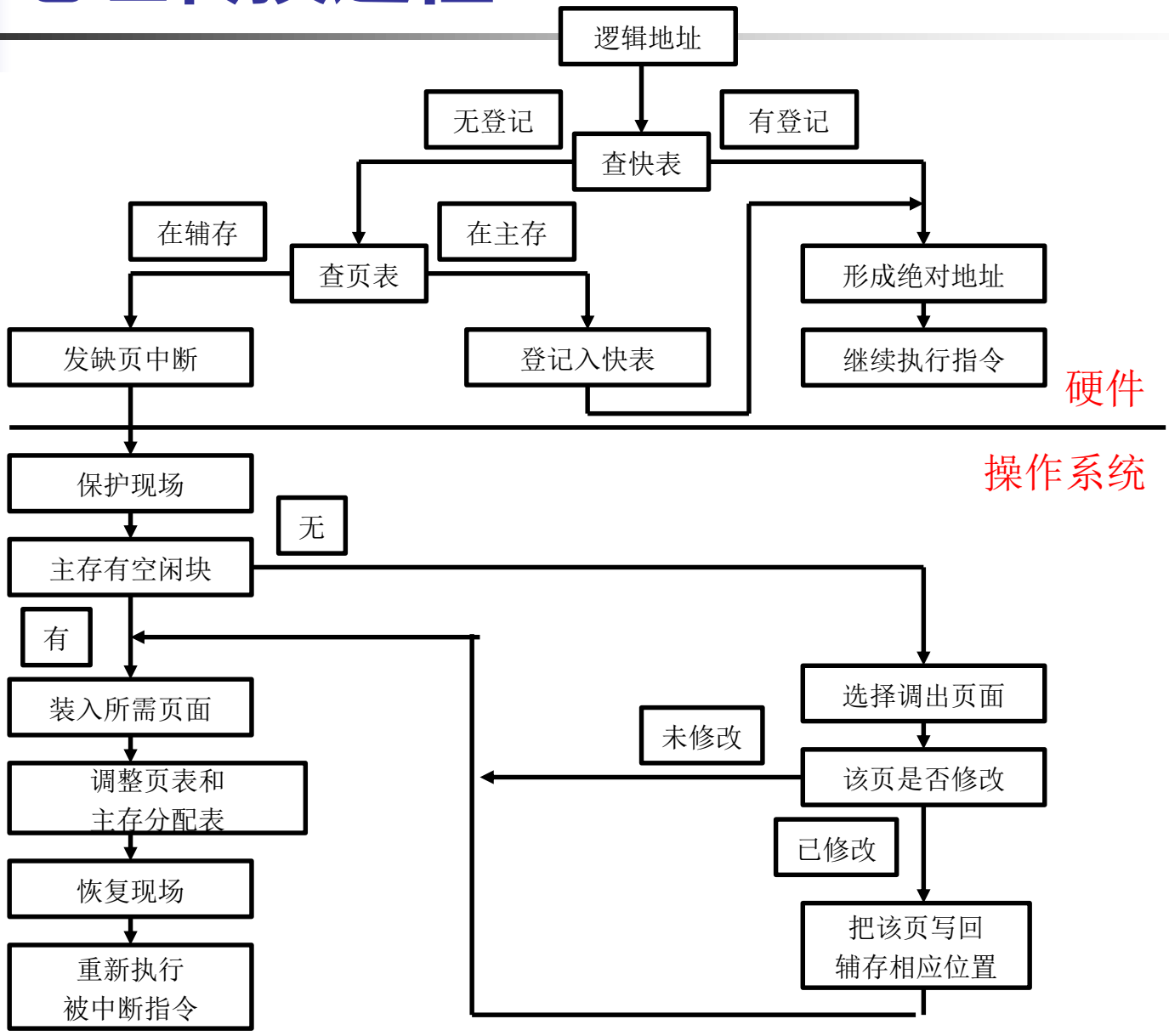
- 请求分页虚拟存储管理系统的地址变换过程类似于分页存储管理

- 区别：但**当被访问页不在内存时**应进行**缺页中断处理**。

请求分页虚存地址转换过程

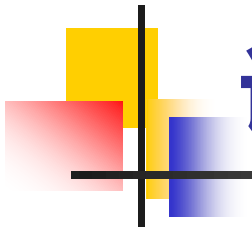


地址转换过程


逻辑地址
无登记
有登记
查快表
在辅存
在主存
查页表
形成绝对地址
发缺页中断
登记入快表
继续执行指令
硬件
操作系统
保护现场
无
主存有空闲块
有
装入所需页面
选择调出页面
未修改
调整页表和
主存分配表
该页是否修改
已修改
恢复现场
把该页写回
辅存相应位置
重新执行
被中断指令

请求页式虚拟存储系统优缺点

- 优点：
  - 作业的程序和数据可按页分散存放在内存中，减少移动开销，有效解决了碎片问题;
  - 既有利于改进主存利用率，又有利于多道程序运行。
- 缺点：
  - 要有硬件支持，要进行缺页中断处理，机器成本增加，系统开销加大。



讨论

- 假设程序刚引用了一个虚存地址，描述以下的可能场景
  - TLB未命中，没有缺页错误
  - TLB未命中，有缺页错误
  - TLB命中，没有缺页错误
  - TLB命中，有缺页错误

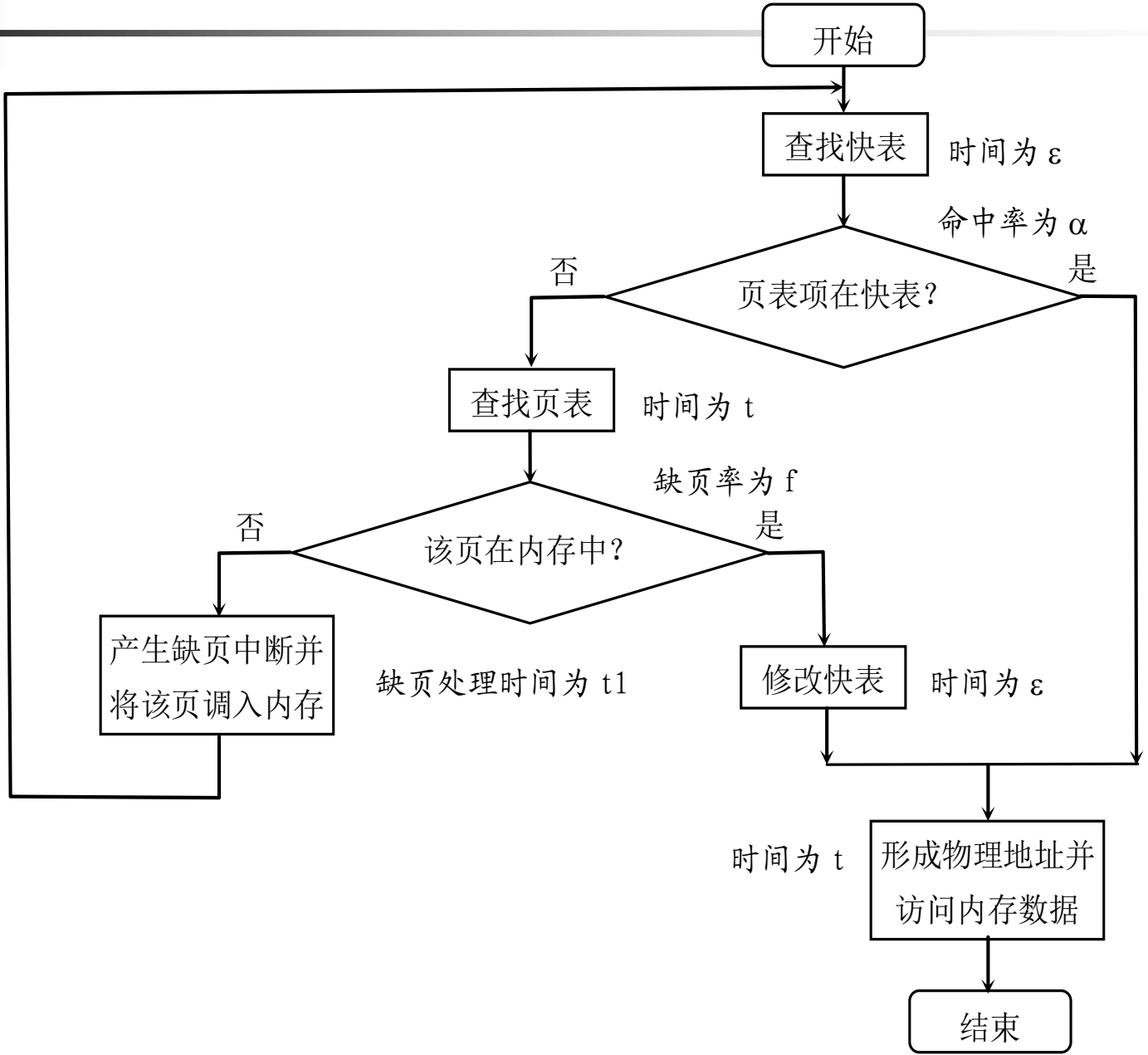
TLB的刷新问题

- 为什么要刷新TLB
  - 进程切换
  - 请求调页与页面替换
    - 被替换页的页表项，如果存在于TLB中，则该TLB是脏的
- 现代计算机系统支持对TLB的刷新问题
  - 硬件管理TLB
    - TLB不命中后，CPU自动查找页表并更新TLB
  - 软件管理TLB
    - TLB不命中触发异常，OS负责查找页表并更新TLB

缺页中断中的TLB刷新问题

- 典型的指令集包括TLB相关的特权指令，使OS能对该指令中对TLB的某一行进行更新
  - X86: INVLPG，只是将TLB设置为invalid，不更新TLB
  - MIPS：LDTLB，更新TLB
- 缺页中断处理程序是否同时更新TLB?
  - 看不同的软硬件环境要求描述

请求分页虚存储系统内存访问的评价



访内存操作情况

- 若页在主存中且页表项在快表中：
  访问时间=查快表时间+访问内存时间=ε+t。

- 若页在主存中且页表项不在快表中：
  - 访问时间=查快表时间+查页表时间+修改快表时间+访问内存时间=ε+t+ε+t=2(ε+t)

- 若页不在主存中，处理缺页中断的时间为tl(包含读入缺页、页表更新、快表更新时间)：
  - 访问时间=查快表时间+查页表时间+处理缺页中断时间tl+查快表时间+访问内存时间
    =ε+t+t1+ε+t=tl+2(ε+t)

有效访问时间

- “快表”的命中率为α，缺页中断率为f，则有效存取时间可表示为：
  - EAT= α *(ε +t)+(1- α)*
    [(1-f) *2 (ε +t) +f*(tl+ 2 (ε +t))]

请求分页的优化

- 交换空间的处理速度远远快于文件系统I/O
  - 因为：交换空间按大块进行分配，不需要文件系统中的查找、间接分配等复杂管理
- 策略一：在进程装入时，将整个进程映像拷贝到交换分区
  - 然后在交换空间执行换入、换出
  - 被老版本的BSD采用
- 策略二：刚开始时，从文件系统进行请求调页，在替换页面时将页面写入交换分区
  - 保证只从文件系统读取所需的页面
  - 后续调页都从交换空间完成，减少文件系统I/O，提高速度

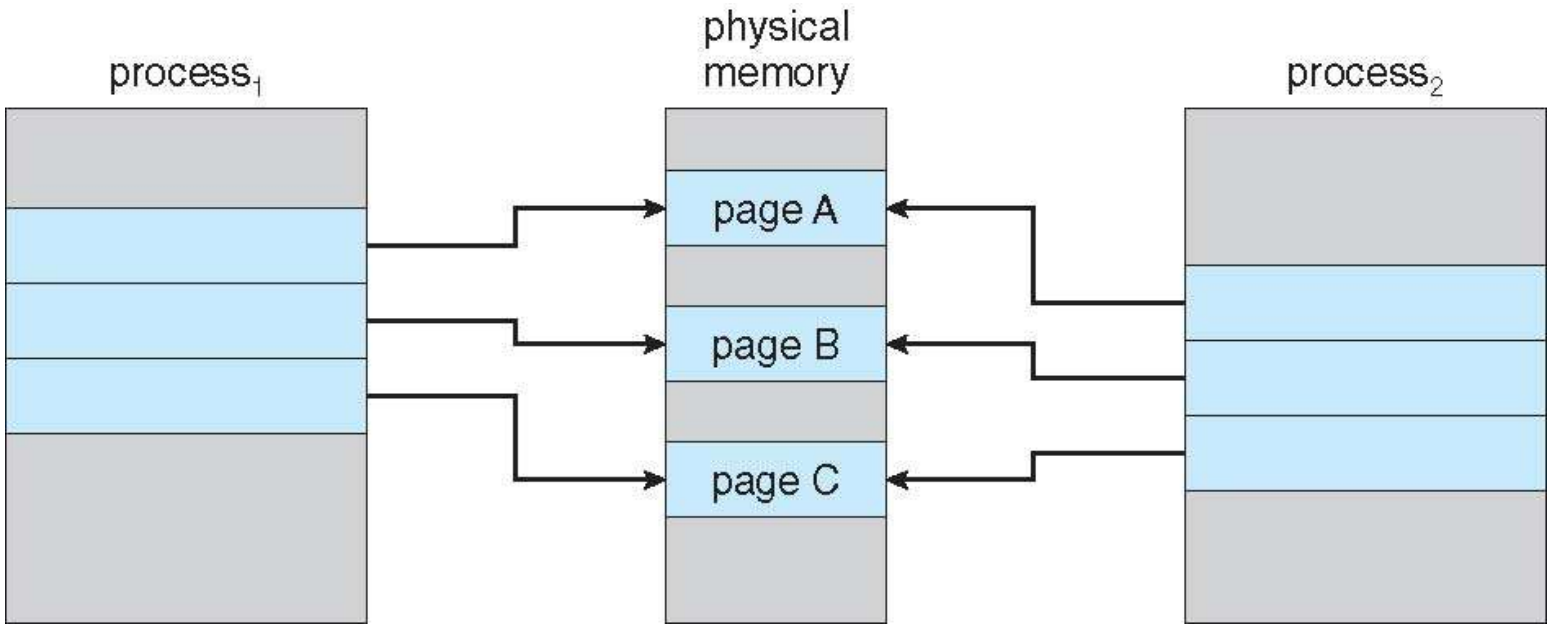
请求分页的优化(Cont.)

- 对于二进制文件的请求调页
  - 直接从文件系统读取，当置换时，直接丢弃（因为只读）
  - 再次需要时，直接从文件系统读取
- 与文件无关的页面仍然需要交换分区
  - 与文件无关的：stack & heap，称为匿名内存
  - BSD Solaris仍然采用
- 移动操作系统
  - 典型的系统不支持交换，如Android，iOS
  - 请求调页：直接从文件系统调入，只回收只读页面

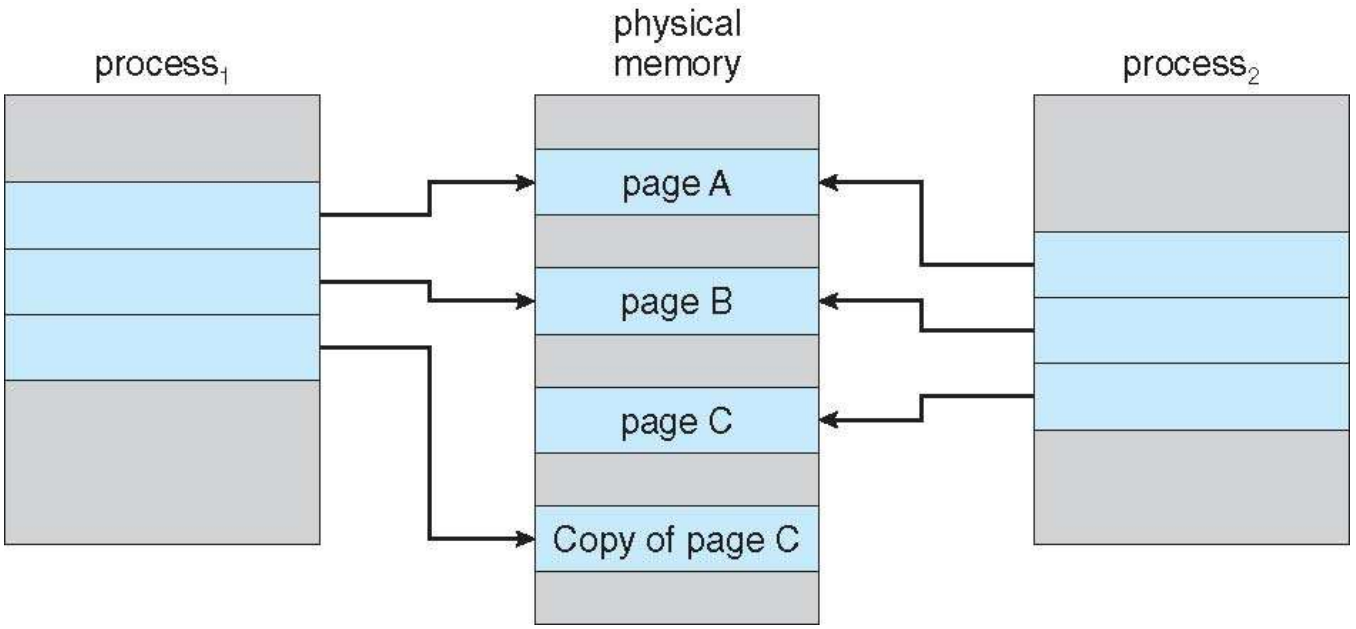
7.3 Copy-on-Write

- Copy-on-Write (COW) 允许父子进程最开始共享相同的内存页面
  - 如果任何一个进程修改了共享页面，则页面将被复制
- COW提供了快速的进程创建，允许最小化的创建新进程必须分配的新页面数量
- 当一个页面被确定要采用写时复制时，操作系统按照zero-fill-on-demand技术，从缓冲池分配页面
  - 预先设置一个空闲的页面池
  - 分配之前先填零，从而清除以前的内容
- vfork()：virtual memory fork
  - fork()的变种，但不采用COW
  - 父进程被挂起，子进程使用父进程的地址空间
  - 子进程的页面修改对父进程时可见的，使用时要尽量避免修改父进程
  - 被设计的使用场景为：子进程创建后，直接调用exec()，而非修改父进程地址空间
  - 无页面复制，更高效

Process 1 修改 Page C前



Process 1 修改 Page C后



7.4 页面替换算法

- 页面替换需要考虑的问题：
  - 避免内存的过度分配
    - 如：6个进程，10页面/进程，但实际访问5页/进程，若全部分配，则浪费30个物理页，若有临时额外追加10页访问，则只需40页
    - 无用的页面可以替换
  - 使用修改位(脏位)减少页面传输代价
    - 只需要将修改的页面写回磁盘，其余丢弃。
- 页面替换能够完全分离逻辑内存和物理内存，实现在小物理内存上提供大的虚拟内存
- 页面替换算法又称为页面淘汰算法，是用来选择换出页面的算法。

1. 基本的页面替换原则

① 在磁盘上找到所需页面的位置

② 寻找空闲的物理页帧

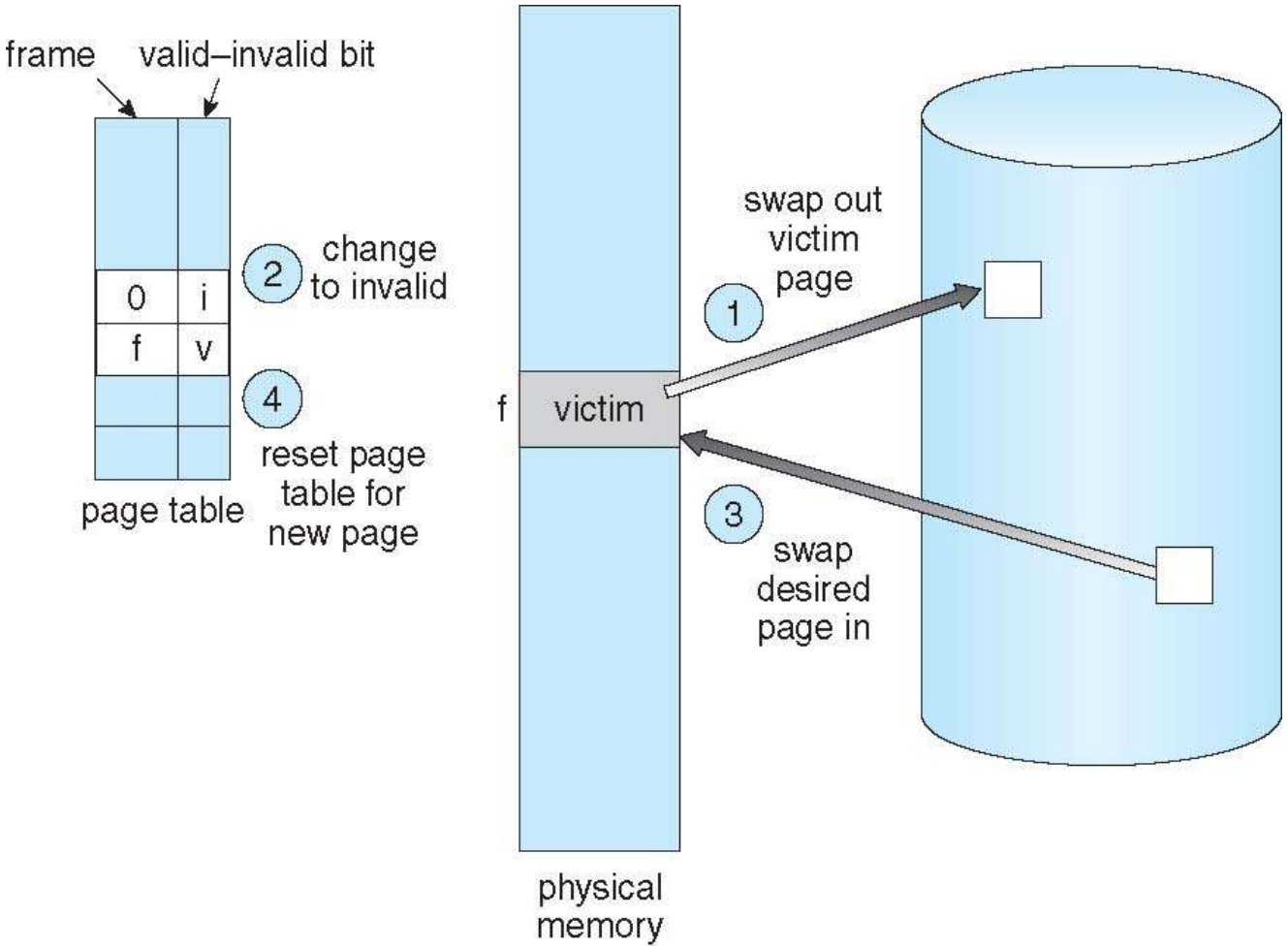
- 如果找到了空闲页帧，使用它
- 如果没有找到，则使用页面替换算法选择一个已经被使用的页帧，作为牺牲页面(victim frame)
  - 如果该牺牲页面是脏的，则要写回磁盘

③ 将所需页面写到新的空闲页帧中，更新页表和页帧表

④ 重启用户进程

- 注意：当没有空闲页帧时，两个页面的传输会增加有效访问时间。*是哪2个页面呢?*

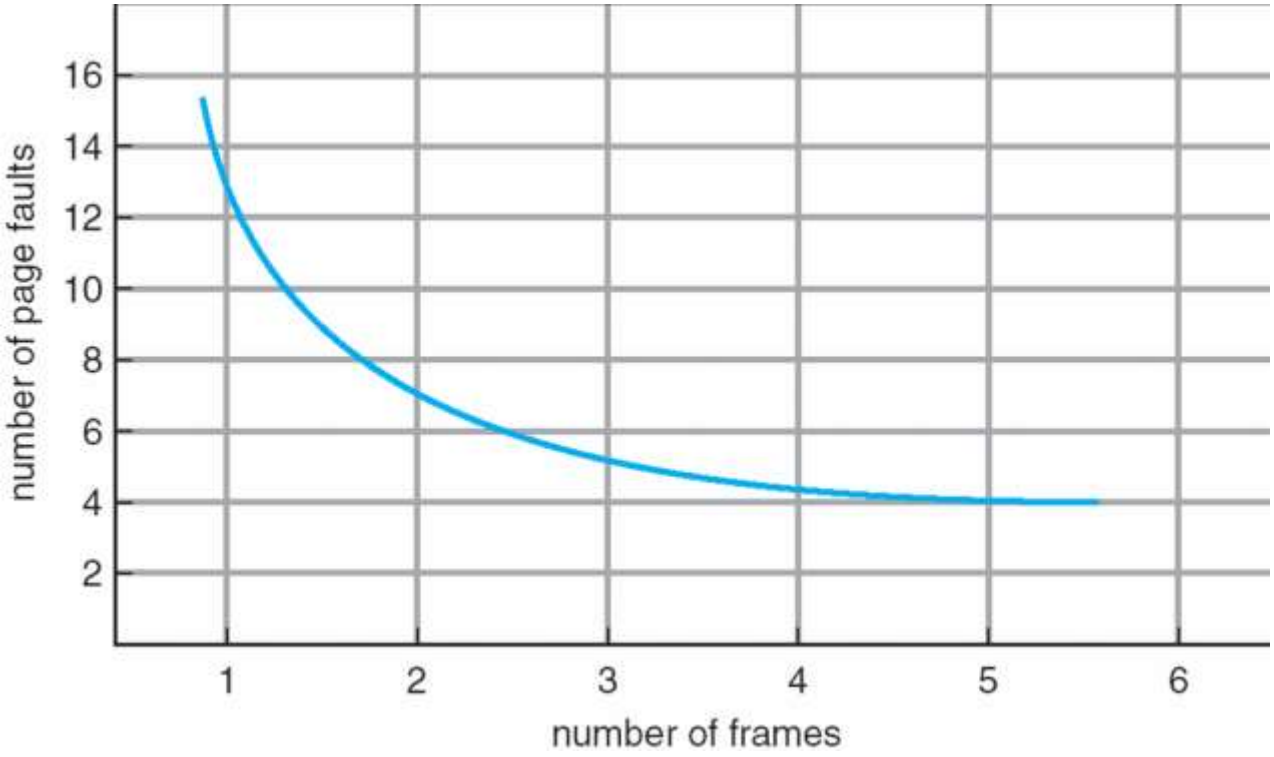
页面替换原理图


frame
valid–invalid bit
0 i
f v
page table
2
change to invalid
4
reset page table for new page
f
victim
1
swap out victim page
3
swap desired page in
physical memory

页面替换算法和页帧分配算法

- 请求调页需要解决的两个问题
  - 页帧分配算法(frame allocation)
  - 页面替换算法(page replacement)
  - 要给每个进程多少物理页帧
  - 当需要页面替换时候，需要决定哪一个页帧被替换
- 页面替换算法
  - 要考虑当本次访问和再次访问时，确保最小的缺页故障率
- 评估算法
  - 给定一个特定的内存引用串（reference string），计算缺页故障率
    - 只需要考虑页号，而不需要完整地址
    - 当对一个页 p访问，则紧跟着的访问p页面请求不会引起Page Fault

缺页故障和页帧数量的关系



2. 先进先出算法（FIFO）

- 前提假设：
  - 程序是按线性顺序来访问物理空间的
- 核心思想：
  - 选择调入主存时间最长的页面予以淘汰，认为驻留时间最长的页面不再使用的可能性较大。

FIFO例1：无异常现象的页面序列

- 假定系统为某进程分配了3个物理块，页面访问序列为：7、0、1、2、0、3、0、4、2、3、0、3，开始时3个物理块均为空闲，采用先进先出替换算法时的页面替换情况如下所示：

走向	7	0	1	2	0	3	0	4	2	3	0	3
块1	7	7	7	2		2	2	4	4	4	0	
块2		0	0	0		3	3	3	2	2	2	
块3			1	1		1	0	0	0	3	3	
	缺	缺	缺	缺		缺	缺	缺	缺	缺	缺	

- 共发生了10次缺页中断。其缺页率为10/12＝83.3% 。

为进程分配4个物理块

走向	7	0	1	2	0	3	0	4	2	3	0	3
块1	7	7	7	7		3		3			3	
块2		0	0	0		0		4			4	
块3			1	1		1		1			0	
块4				2		2		2			2	
	缺	缺	缺	缺		缺		缺			缺	

- 从上表中可以看出，共发生了7次缺页中断。其缺页率为7/12 = 58.3% 。

FIFO例2：有异常现象的页面序列

- 假定系统为某进程分配了3个物理块，页面访问序列为：1、2、3、4、1、2、5、1、2、3、4、5，开始时3个物理块均为空闲，采用先进先出替换算法时的页面替换情况如下所示：

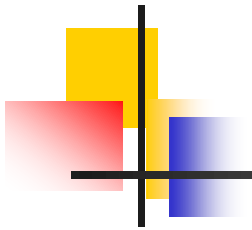
走向	1	2	3	4	1	2	5	1	2	3	4	5
块1	1	1	1	4	4	4	5			5	5	
块2		2	2	2	1	1	1			3	3	
块3			3	3	3	2	2			2	4	
	缺	缺	缺	缺	缺	缺	缺			缺	缺	

- 共发生了9次缺页中断。其缺页率为9/12＝75% 。

为进程分配4个物理块

走向	1	2	3	4	1	2	5	1	2	3	4	5
块1	1	1	1	1			5	5	5	5	4	4
块2		2	2	2			2	1	1	1	1	5
块3			3	3			3	3	2	2	2	2
块4				4			4	4	4	3	3	3
	缺	缺	缺	缺			缺	缺	缺	缺	缺	缺

- 共发生了10次缺页中断。其缺页率为10/12 = 83.3% 。
- 分配的物理页框增加了，缺页率反而升高了！



- FIFO算法特点：
  - 实现比较简单
  - 对按线性顺序访问的程序比较合适，对其他特性的程序则效率不高，Why?
  - 但可能产生异常现象：
    - Belady 现象：在某些情况下，分配给进程的页面数增多，缺页次数反而增加。
    - 原因：FIFO算法的置换特征与进程访问内存的动态特征是矛盾的，即被置换的页面并不是进程不会访问的，因而FIFO并不是一个好的置换算法。
    - Belady现象与抖动现象是不完全相同的。
  - 很少使用纯粹的FIFO

3. 最佳替换算法(OPT)

- Optimal replacement, OPT
- 核心思想：淘汰掉将来不再访问，或者距现在最长时间后才可能会访问的页面，
- 换言之是从内存中选择将来最长时间不会使用的页面予以淘汰。
- 缺点：
  - 因页面访问的未来顺序很难精确预测
  - 该算法具有理论意义，可以用来评价其他算法的优劣。

最佳替换算法例

- 假定系统为某进程分配了3个物理块，页面访问序列为：1、2、3、4、1、2、5、1、2、3、4、5，开始时3个物理块均为空闲，采用最佳替换算法时的页面替换情况如下所示：

走向	1	2	3	4	1	2	5	1	2	3	4	5
块1	1	1	1	1			1			3	3	
块2		2	2	2			2			2	4	
块3			3	4			5			5	5	
	缺	缺	缺	缺			缺			缺	缺	

- 共发生了7次缺页，其缺页率为7/12＝58.3% 。

4.最近最久未使用替换算法(LRU)

- Least Recently Used:
  - 基于局部性原理：刚被使用过的页面可能还会立即被使用，较长时间内未被使用的页面可能不会立即使用。
  - 选择最近一段时间内最长时间未被访问过的页面予以淘汰。

- 应赋予每个页面一个访问字段，用于记录页面自上次访问以来所经历的时间，同时维护一个淘汰队列

LRU算法例

- 假定系统为某进程分配了3个物理块，页面访问序列为：1、2、3、4、1、2、5、1、2、3、4、5，开始时3个物理块均为空闲，采用LRU替换算法时的页面替换情况如下所示：

走向	1	2	3	4	1	2	5	1	2	3	4	5
块1	1	1	1	4	4	4	5			3	3	3
块2		2	2	2	1	1	1			1	4	4
块3			3	3	3	2	2			2	2	5
	缺	缺	缺	缺	缺	缺	缺			缺	缺	缺

- 从上表中可以看出，共发生了10次缺页，其缺页率为10/12＝83.3% 。

为进程分配4个物理块

走向	1	2	3	4	1	2	5	1	2	3	4	5
块1	1	1	1	1			1			1	1	5
块2		2	2	2			2			2	2	2
块3			3	3			5			5	4	4
块4				4			4			3	3	3
	缺	缺	缺	缺			缺			缺	缺	缺

- 共发生了8次缺页中断。其缺页率为8/12 = 67.7% 。

LRU算法的实现

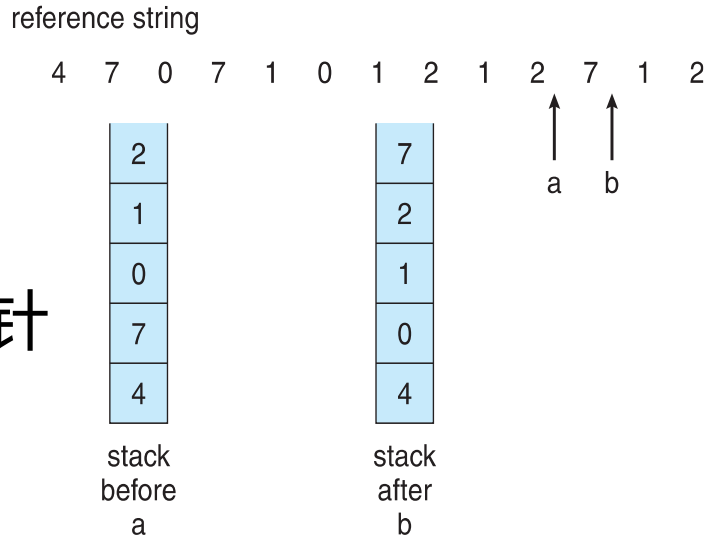
- LRU算法实现时需要较多的硬件支持，以记录进程中各页面自上次访问以来有多长时间未被访问，主要实现方法有：
  ① 计数器
  ② 堆栈

① 计数器法

- 主要思想
  - 为每个页表项关联一个时间域字段
  - 为CPU增加一个硬件计数器或者逻辑时钟
  - 使用硬件计数器自动计数，每当访问一页时，计数器自动加１，并将计数器值复制到相应页所对应页表项的时间域内。
  - 每隔t时间，对计数器自动清零。
  - 当发生缺页中断时，可选择计数值最小的对应页面淘汰，并将所有计数器全部清０。

② 堆栈法

- 堆栈实现
  - 用一个具有头指针和尾指针的双向链表来实现一个堆栈
  - 每当引用一个页
    - 将该页从栈中移至栈顶部
    - 栈顶中记录了最近使用的页号
    - 栈底部是需要替换的页面
  - 每次更新需要调整栈中每项的指针
  - 替换的选择不需要搜索。



- OPT和LRU都称为堆栈算法，没有belady现象
  - Why?
  - 但是：这两种算法需要硬件支持，否则，如果在中断中以软件方式修改这些计时记录，会非常慢，10倍以上代价！

5. LRU算法近似算法

- LRU算法需要足够的硬件支持，且仍然很慢
- 变通：引用位法
  - 每一个页面关联一个bit，初始为0
  - 当页面被引用时，设置为1
  - 需要替换页面时，替换掉bit=0的页面（如果存在）
    - 虽然不了解访问顺序，但是了解哪些没被访问过

（1）附加引用位法

- 每页都有引用位，并为每页设一个8位内存信息
- 每隔规定时间（如100ms），时钟定时器触发中断，将控制权交给OS
- OS将每个页的引用位转移到8位的高位，并将其他位右移1位，抛弃最低位
- 这8位就表明了最近8个时间周期，页面的使用情况
- 发生缺页时，挑选最小的为LRU页替换
- 在这里，最高位是引用位，其他位为历史位

页面 \ R	R7	R6	R5	R4	R3	R2	R1	R0
1	0	1	0	1	0	0	1	0
2	1	0	1	0	1	1	0	0
3	1	0	0	0	0	0	0	0
4	0	1	1	0	1	0	1	1
5	1	1	0	1	0	1	1	0
6	0	0	1	0	1	0	1	1
7	0	0	0	0	0	1	1	1
8	0	1	1	0	1	1	0	1

(2) 二次机会算法

- 当把历史位数从7设置为0，那么只剩下引用位，则算法退化为二次机会算法

- 二次机会算法(Second Chance Replacement，SCR)是FIFO算法的改进，以避免将经常使用的页面淘汰掉，为此设置了页面引用位。

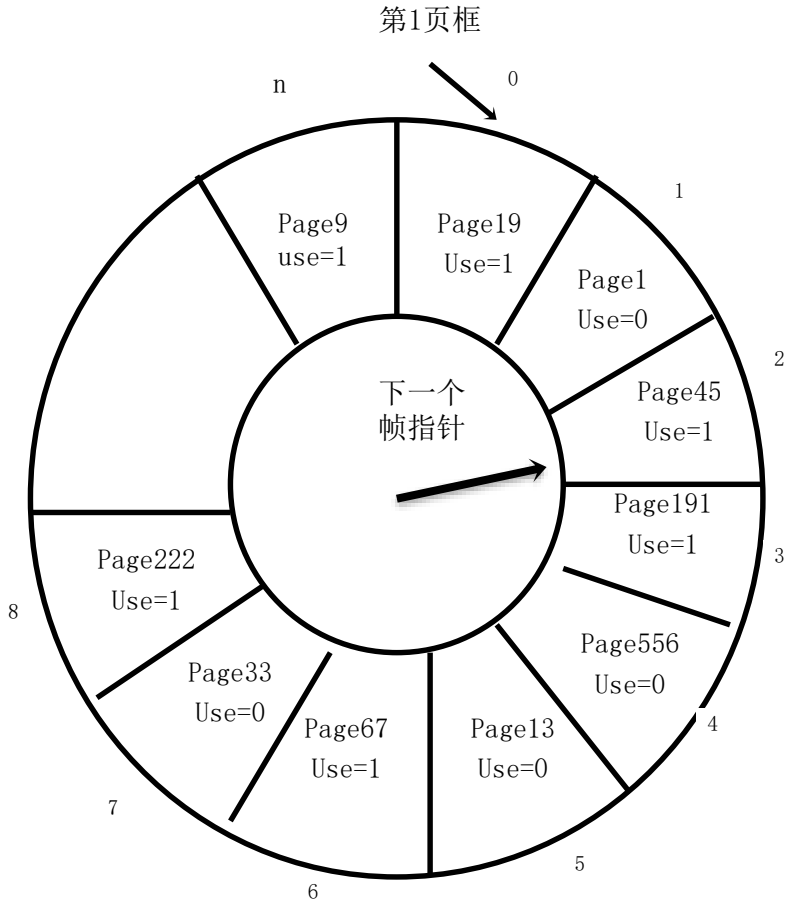
- 算法思想：使用FIFO算法选择一页淘汰时，先检查该页的引用位
  - 如果是0就立即淘汰该页
  - 如果是1就给它第二次机会，将其引用位清0，并将它放入页面链的末尾，将其装入时间置为当前时间，然后选择下一个页面。

（3）简单时钟（clock）算法

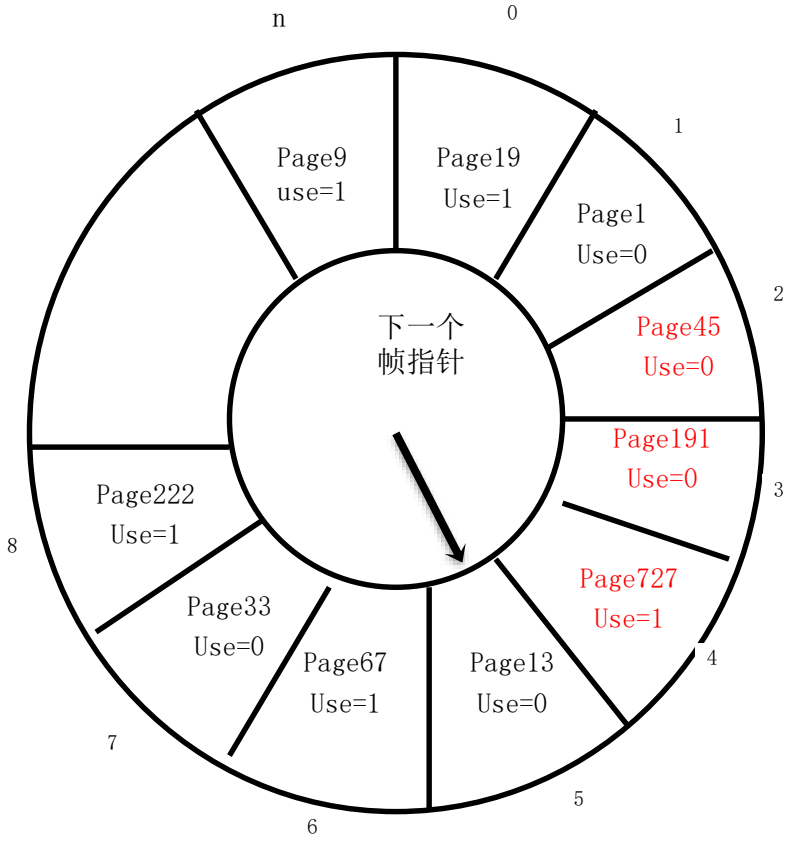
- 简单时钟替换算法是对二次机会算法的一种循环队列实现

- 实现思想：将页面排成一个循环队列，类似于时钟表面，并使用一个替换指针。
  - 当发生缺页时，检查指针指向的页面
  - 若其访问位为0，即找到牺牲页，则淘汰该页
  - 否则将该页的访问位清0，指针前移并重复上述过程，直到找到牺牲页为止。
  - 最后指针停留在牺牲页的下一页上；

时钟策略的一个例子

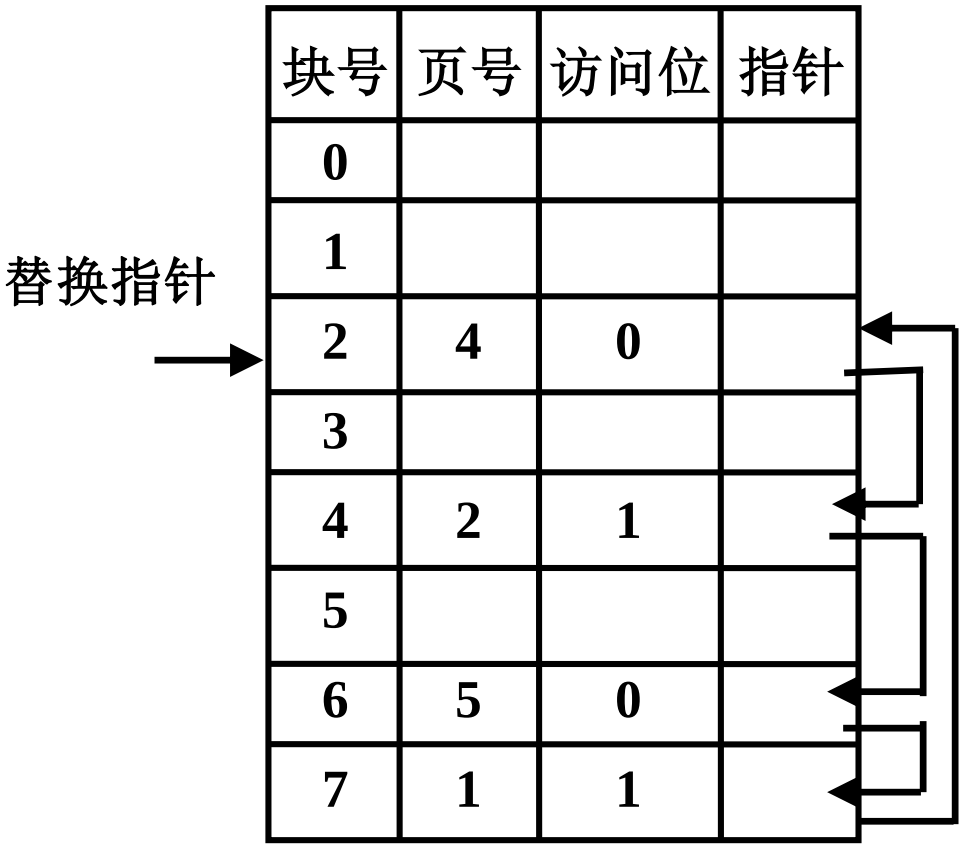


一个页替换前的缓冲区状态



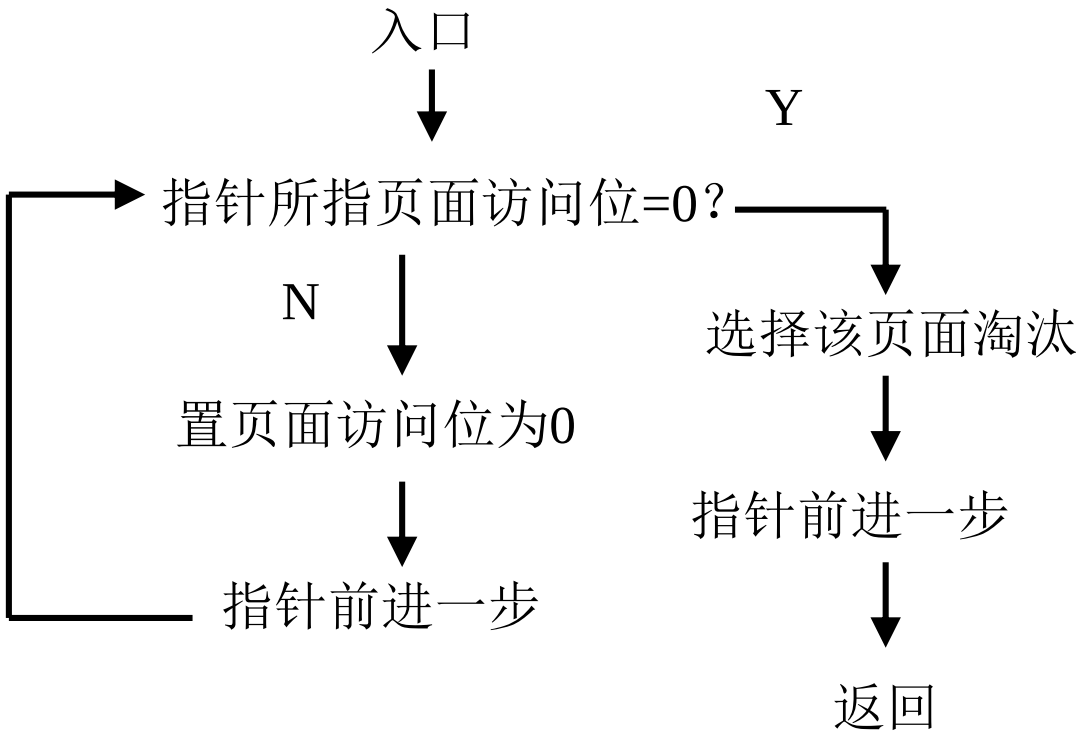
下一页替换后的缓冲区状态

简单时钟替换算法页面链



块号	页号	访问位	指针
0			
1			
2	4	0	
3			
4	2	1	
5			
6	5	0	
7	1	1	

简单时钟替换算法流程

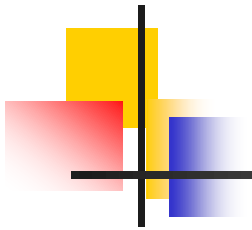


简单时钟替换例

此时为何块2、块3均无*标记？

走向	7	0	1	2	0	3	0	4	2	3	0	3
块1	7*	7*	7*	2*	2*	2*	2*	4*	4*	4*	4	3*
块2		0*	0*	0	0*	0	0*	0	2*	2*	2	2
块3			1*	1	1	3*	3*	3	3	3*	0*	0*
	缺	缺	缺	缺		缺		缺	缺		缺	缺

- 从上表中可以看出，共发生了9次缺页中断。其缺页率为9/12 = 75% 。



- 在这里要注意，简单时钟，实际上是一个FIFO的改进，即二次机会的另外一种实现形式，因此，对于那些已经进入引用的页面，他们不影响当前指针的情况，即不是最近使用过的就需要调整。

- 算法的极端情况：
  - 所有页都被引用，则选择时，需要先把所有页面遍历，清除所有引用位，则退化为FIFO

（4）增强的时钟算法

- 问题：将一个修改过的页面换出需要写磁盘，其开销大于未修改页面，为此在改进型时钟算法中应考虑页面修改情况，将访问情况与修改情况结合使用。

- 设R为访问位/引用位，M为修改位，将页面分为以下4种类型：
  - R=0，M=0：未被访问又未被修改——用于置换的最佳页面
  - R=0，M=1：未被访问但已被修改——若置换需写回
  - R=1，M=0：已被访问但未被修改——可能很快被再次访问
  - R=1，M=1：已被访问且已被修改——可能很快被再次访问且要写回

算法描述

- 步骤1：从指针当前位置开始扫描循环队列，寻找R=0，M=0的页面，将满足条件的第一个页面作为淘汰页，本轮扫描不修改“访问位R”。（若失败，则R=0,M=1; R=1, M=0/1）

- 步骤2：若第1步失败，则开始第2轮扫描，寻找R=0，M=1的页面，将满足条件的第一个页面作为淘汰页，并将所有经历过页面的访问位R置0。（若失败，R=1,M=0/1）

- 步骤3：若第2步失败，则将指针返回到开始位置，然后重复第1步，若仍失败则必须重复第2步，此时一定能找到淘汰页面。

- 特点：
  - 被称为“第三次机会时钟替换算法”。
  - 减少了磁盘I/O次数，但算法本身开销增加。

6. 基于计数的页替换算法

- 为每页保留一个引用计数器
- LFU (最不常使用页替换):
  - 观点：替换计数最小的页面
  - 一个问题：
    - 可能有的页面一段时间获得高count值后，较长时间不再访问
    - 解决：将计数寄存器右移，形成指数衰减的平均使用计数
- MFU(最常使用页替换)：
  - 观点：最小计数的页面可能刚刚调入
- LFU和MFU都不常用且费时

7. 页面缓冲算法

- 系统通常保留一个空闲页帧缓冲池
  - 当发生page fault时，仍然选择牺牲帧
  - 将所需的页面信息读入空闲帧，然后再选择牺牲帧驱逐
  - 当需要时，驱逐牺牲帧，再将牺牲帧加入空闲缓冲池
  - 这种方法不需等待牺牲帧的写回磁盘，提高进程重启速度
- 扩展一：保留一个修改页列表
  - 当调页设备空闲时，选择一个修改页面并写回，设置该页为干净页面
  - 提高了选择替换时，选到干净页的概率
- 扩展二：保留一个空闲帧池，并记住页和帧的对应关系
  - 如果有页面再次被访问，则不需从磁盘上装入
  - 通常，对于解决算法错误地选择牺牲帧的问题非常有效

8.应用程序和页面替换

- 前面的算法都是假设操作系统对未来页面访问的猜测是准确的
- 但是，某些应用程序有更好的领域知识
  - 如数据库
- 内存密集型应用程序会带来两级缓存，矛盾：
  - OS维护了I/O缓冲时内存页面，LRU会删掉旧的页面访问
  - DB维护了其自身工作时的内存页面，顺序磁盘访问，需要访问旧页面，MFU更好
- 改进：操作系统提供直接的磁盘访问
  - Raw disk mode
  - 绕过了缓冲、文件锁、文件系统等

7.5 页面管理的策略问题

1. 页面装入与清除策略

2. 页面的分配与替换策略

1、页面装入和清除策略策略

- 页面装入策略：决定何时把页面装入主存
- 有两种策略：
  - 请页式调度(demand paging)
    - 按需装入
    - 频繁磁盘I/O
  - 预调式调度(prepaging)
    - 局部性原理，动态预测，预先装入；
    - Windows代码页面预调3-8页，数据页面2-4页

页面装入和清除策略策略

- 页面清除策略：决定何时把一个修改过的页面**写回辅存储器**
- 有两种策略：
  - 请页式清除
    - 当一页被选中进行替换，且被修改过时，则进行写回磁盘
    - 缺点：效率低下
  - 预约式清除
    - 对所有修改的页面，替换前，提前成批写回
    - 要写回的页仍然在主存，直到被替换算法选中此页从主存中移出。
    - 若该页面在刚被写回后，在替换前，再次被大量修改，则该策略失效。

2、页面分配和替换策略

- **如何为进程分配物理页框?**
  - 进程需要的最少物理块数?
  - 进程的物理块数是固定还是可变?
  - 按什么原则为进程分配物理块数?

(1)最小物理块数的确定

- 最小物理块数是指能保证进程正常运行所需的最少物理块数，若小于此值，进程无法运行。
- 一般情况下：
  - 单地址指令且采用直接寻址，则所需最少物理块数为2
  - 若单地址指令且允许采用间接寻址，则所需最少物理块数为3。
  - 某些功能较强的机器，指令及源地址和目标地址均跨两个页面，则最少需要6个物理块。

(2)页面分配策略：固定分配与可变分配

- 固定分配
  - 进程保持页框数固定不变，称固定分配;
  - 进程创建时，根据进程类型和程序员的要求决定页框数，只要有一个缺页中断产生，进程就会有一页被替换。

- 可变分配
  - 进程分得的页框数可变， 称可变分配;
  - 进程执行的某阶段缺页率较高，说明目前局部性较差，系统可增加分配页框以降低缺页率，反之说明进程目前的局部性较好，可减少分配进程的页框数

(3)页面替换策略：局部替换和全局替换

- 如果页面替换算法的作用范围是整个系统，称全局页面替换算法，它可以在运行进程间动态地分配页框。
- 如果页面替换算法的作用范围局限于本进程，称为局部页面替换算法，它实际上需要为每个进程分配固定的页框。
- 工作集：为每个进程所维护的一组页面
  - 会自动排除不再在工作集中的页面，因此工作集会随着进程的执行而变化
  - 若进程的工作集大小在运行时变化较大的，全局替换比局部替换策略更优，但为每个进程确定的分配页框数是困难的

(4)分配和替换算法的配合

- 固定分配+局部替换
- 可变分配+全局替换
- 可变分配+局部替换

固定分配+局部替换

- 进程分得的页框数不变，发生缺页中断，只能从进程的页面中选页替换，保证进程的页框总数不变。

- 策略难点：应给每个进程分配多少页框?
  - 给少了，缺页中断率高;
  - 给多了，使主存中能同时执行的进程数减少，进而造成处理器和其它设备空闲。

- 采用固定分配算法，系统把页框分配给进程，可采用一些策略，如:
  - 平均分配
  - 按比例分配
  - 优先权分配

可变分配+全局替换

- 实现要点
  - 每个进程分配一定数目页框，OS保留若干空闲页框。
  - 进程发生缺页中断时，从系统空闲页框中选一个给进程，这样产生缺页中断进程的主存空间会逐渐增大，有助于减少系统的缺页中断次数。
  - 系统拥有的空闲页框耗尽时，会从主存中选择一页淘汰，该页可以是主存中任一进程的页面，这样又会使那个进程的页框数减少，缺页中断率上升。

- 策略难点：如何选择哪个页面作为替换？
  - 应用某种淘汰策略选择页面，并未确定哪个进程会失去页面
  - 如果选择某个进程，此进程工作集合的缩小会严重影响其运行，那么这个选择就不是最佳的了。

可变分配+局部替换

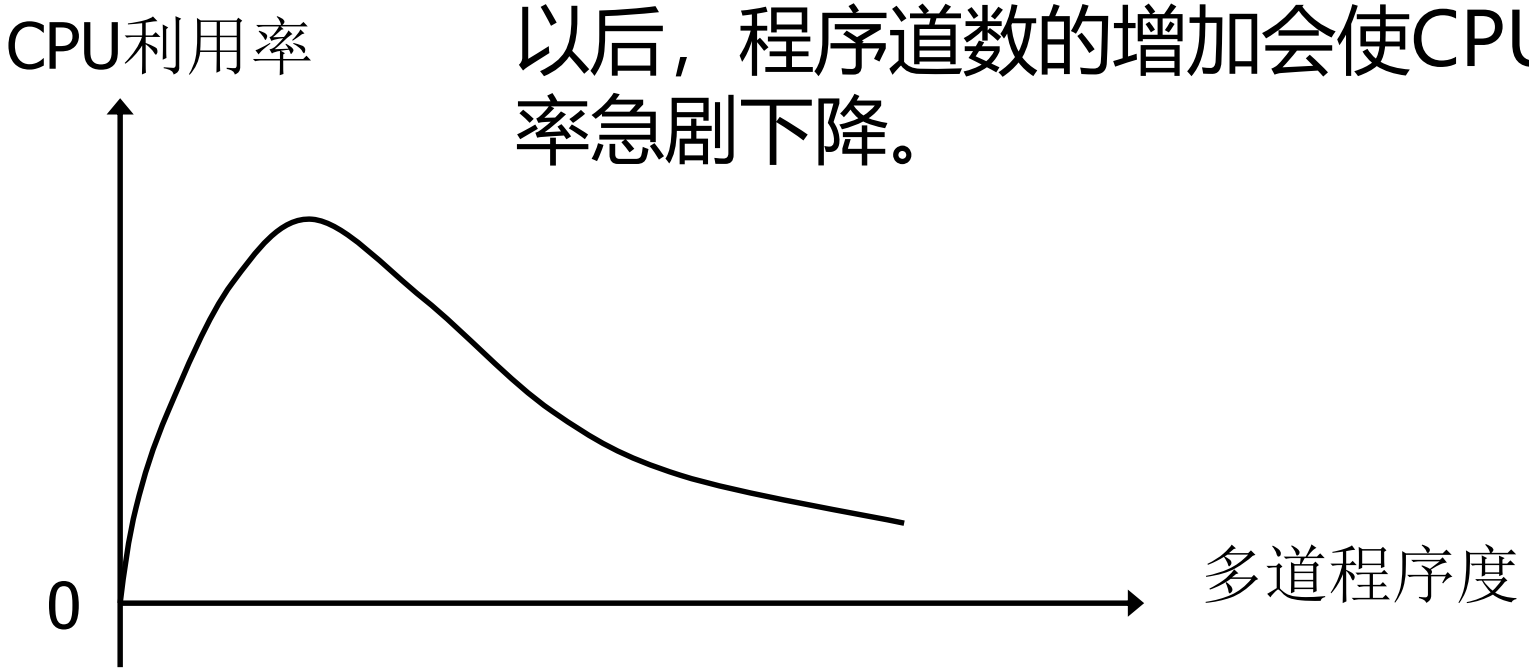
- 实现要点
  - 新进程装入主存时，根据应用类型、程序要求，分配给一定数目页框，可用请页式或预调式完成这个分配。
  - 产生缺页中断时，从该进程驻留集中选一个页面替换。
  - 不时重新评价进程的分配，增加或减少分配给进程的页框以改善系统性能。

7.6 抖动问题

- 如果运行进程的大部分时间都用于页面的换入/换出，而几乎不能完成任何有效的工作，则称此进程处于抖动状态。抖动又称为颠簸、颤动。
- 抖动分为：
  - 局部抖动
  - 全局抖动

CPU利用率与多道程序度的关系

- 左图是CPU利用率与程序道数的关系。
- 开始时CPU利用率会随着程序道数的增加而增加，当程序道数增加到一定数量以后，程序道数的增加会使CPU的利用率急剧下降。



抖动产生的原因

- 抖动产生的原因有：
  ① 进程分配的物理块太少

  ② 替换算法选择不当

  ③ 全局替换使抖动传播

抖动的发现

- 抖动发生前会出现一些征兆，可利用这些征兆发现抖动并加以防范。这些技术有：
  ① 全局范围技术
  ② L=S准则
  ③ 利用缺页率发现抖动
  ④ 平均缺页频率

1、全局范围技术

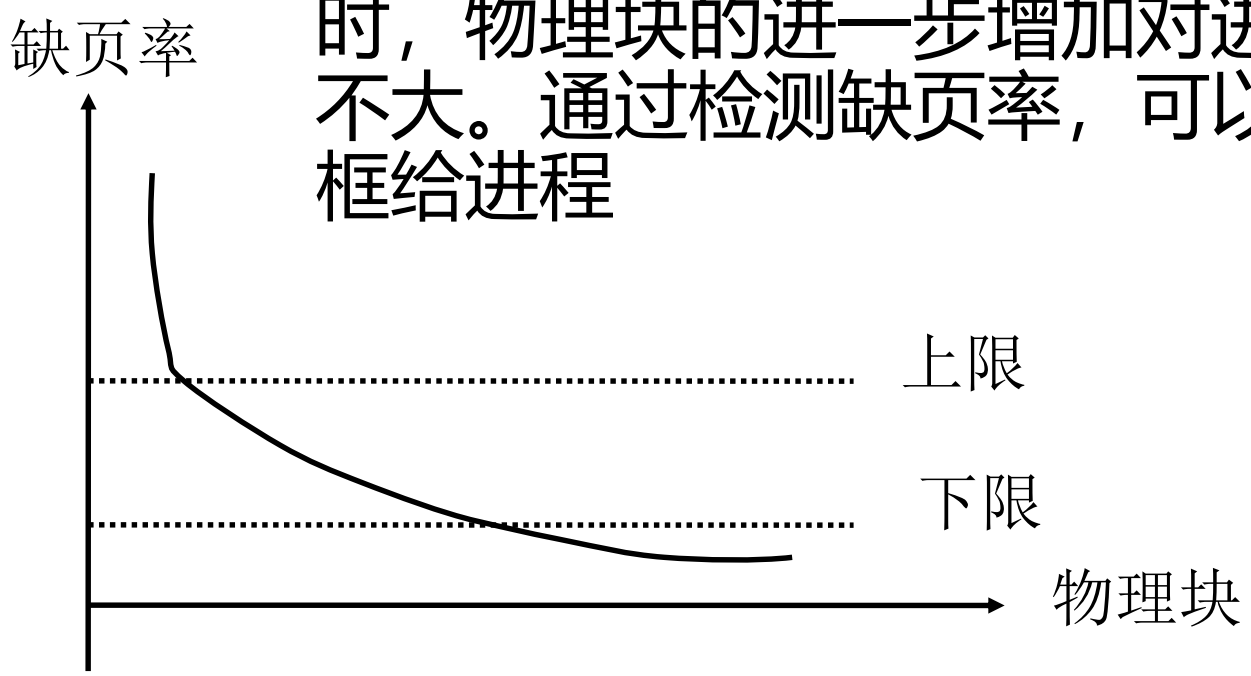
- 全局范围技术采用时钟替换算法，用一个计数器C记录搜索指针扫描页面缓冲的速度。
  - 若C的值大于给定的上限值，说明缺页率太高（可能抖动）或找不到可供替换的页面，这时应减少程序道数。
  - 若C小于给定的下限值，表明缺页率小或存在较多可供替换的页面，这时应增加程序的道数。

2、L=S准则

- 实际情况说明，产生缺页的平均时间L等于系统处理缺页的平均时间S时，CPU的利用率达到最大。
- 当L<S时，表明系统频繁缺页，CPU利用率低，会导致系统抖动。

3、利用缺页率发现抖动

- 下图是缺页率与进程分得物理块数之间的关系。当缺页率超过上限时会引起抖动，因此应增加分配给进程的物理块；此时每增加一个物理块，其缺页率明显降低；当进程缺页率达到下限值时，物理块的进一步增加对进程缺页率的影响不大。通过检测缺页率，可以控制是否分配页框给进程



抖动的预防及解除

- 采用局部替换策略可以防止抖动传播

- 通过挂起进程来解除抖动
- 选择挂起进程的条件
  - 优先级最低：符合进程调度原则
  - 发生缺页中断的进程：内存不含工作集，缺页时应阻塞
  - 最后被激活的进程：工作集可能不在内存
  - 最大的进程：可释放较多空间

程序结构对缺页率影响例

- 设页面大小为128字节，二维数组为128×128，初始时未装入数据，需将数组初始化为0。**若数组按行存放**，问下述两个程序段的缺页率各为多少？
- 程序1

```
char a[128][128];
for (j=0;j<=127;j++)
for (i=0;i<=127;i++)
    a[i][j]=0;
```

程序2

```
for (i=0;i<=127;i++)
for (j=0;j<=127;j++)
    a[i][j]=0;
```

程序1的缺页次数

- 程序1

```
char a[128][128];
for (j=0;j<=127;j++)
  for (i=0;i<=127;i++)
    a[i][j]=0;
```

- 因数组以行为主存放，页面大小为128字节，故每行占一个页面。
- 程序1的内层循环将每行中的指定列置为0，故产生128次中断。
- 外层循环128次，总缺页次数为128×128。

程序2的缺页次数

- 程序2

```
char a[128][128];
for (i=0;i<=127;i++)
  for (j=0;j<=127;j++)
    a[i][j]=0;
```

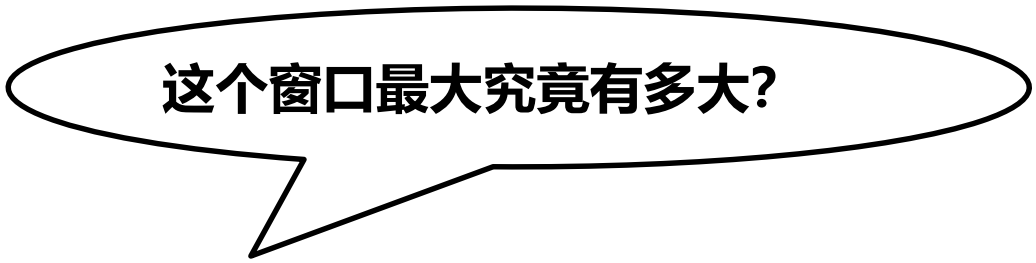
- 因数组以行为主存放，页面大小为128字节，故每行占一个页面。
- 程序2的内层循环将每行的所有列置为0，故产生1次中断。
- 外层循环128次，总缺页次数为128。

7.7 局部页面替换算法

① 局部最佳页面替换算法
② 工作集模型和工作集替换算法
③ 模拟工作集替换算法
④ 缺页频率替换算法

1 局部最佳页面替换算法(Prieve1976)

- 实现思想
  - 分配：进程在时刻t访问某页面，如果该页面不在主存中，导致一次缺页，则把该页面装入一个空闲页框。
  - 替换：不论发生缺页与否，算法在每一次分配时都要考虑引用页面序列，如果该页面在时间间隔(t， t + Δ)内未被再次引用，那么就移出；否则，该页被保留在进程驻留集中。
  - Δ为一个系统常量，间隔(t， t + Δ)称作滑动窗口。



MIN算法例子(Δ=3，最初P4已装入)

时间	0	1	2	3	4	5	6	7	8	9	10
引用序列	4	3	3	4	2	3	5	3	5	1	4
1										√	
2					√						
3		√	√	√	√	√	√	√			
4	√	√	√	√							√
5							√	√	√		
换入		3			2		5			1	4
换出					4	2			3	5	1
窗口集合	3，4	3，4，2	3，4，2	4，2，3，5	2，3，5	3，5	3，5，1	3，5，1，4	5，1，4	1，4	4

缺页数：5，缺页率50%

2、工作集模型(Working Set)

- 提出动机
  - MIN算法，前向查看页面引用串，看将来
  - P. Denning提出，后向看，看历史
- 进程工作集
  - 在某一段时间间隔内进程运行所需访问的页面集合
- 实现思想
  - WS模型基于局部性原理，根据考察最近时间内主存需求，估计不久将来的内存页框需求。

进程工作集

- W(t，Δ)
  - 在时刻t-Δ到时刻t之间，(t-Δ，t)所访问的页面集合，称为进程在时刻t的工作集。
  - Δ是系统定义的一个常量。
  - Δ称为“工作集窗口尺寸”，可通过窗口来观察进程行为，
  - 工作集中所包含的页面数目称为“工作集尺寸”。

WS算法例，Δ=3，W0={5,4,1}

时间	0	1	2	3	4	5	6	7	8	9	10
引用序列	1	3	3	4	2	3	5	3	5	1	4
1	√	√	√	√						√	√
2					√	√	√	√			
3		√	√	√	√	√	√	√	√	√	√
4	√	√	√	√	√	√	√				√
5	√	√					√	√	√	√	√
换入		3			2		5			1	4
换出			5		1			4	2		
工作集合	5,4,1	5,4,1,3	4,1,3	1,3,4	3,4,2	3,4,2	2，3，4，5	2，3，5	3，5	1，3，5	1，3，4，5

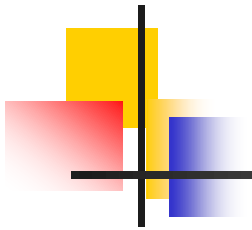
缺页数：5，缺页率50%

通过工作集确定驻留集大小

- 工作集是程序局部性的近似表现
  - 监视每个进程的工作集，只有属于工作集的页面才能留在主存；
  - 定期地从进程驻留集中删去那些不在工作集中的页面；
  - 仅当一个进程的工作集在主存时，进程才能执行。

Windows的页面替换机制

- 结合工作集模型、Clock算法
- 引用页面加入方法
  - 采用局部替换算法，进程缺页时，不会逐出其他进程页面。
  - 工作集最小尺寸(20-50页框)，最大尺寸(45-345页框)；
  - 缺页时，把引用页面添加到进程工作集中，直至达到最大值，若还发生缺页，从工作集中选择淘汰页面，并移出；



- 淘汰页面的选择：
  - 页框有访问位u及计数器count。
  - 该页被引用时，u位被置1；工作集管理程序扫描工作集中页面的访问位，并执行操作：如果u=1，把u和count清0；否则，count加1，扫描结束时，移出count值最大的页面。
- 从工作集中逐出的页框，放入两个主存队列之一
  - 保存暂时移出的并被修改过的页面
  - 保存暂时移出的并为只读的页面

3、模拟工作集替换算法

- 工作集策略在概念上好，但监督驻留页面变化的开销很大，估算合适的窗口Δ大小也是个难题

- 工作集近似算法
  - 老化(Aging)算法（见LRU算法实现部分）
  - 时间戳算法

时间戳算法

- 为页面设置引用位R及关联时间戳timestamp，通过超时中断，每隔若干条指令周期性地检查引用位及时间戳：
  - 当被检查页面的引用位为1，就把它置0，并把这次改变的时间作为时间戳记录下来。
  - 当引用位为0时，用系统当前时间减去时间戳时间，计算出从它上次使用以来未被再次访问的时间量，记入t_off；
- t_off值随着每次超时中断的处理而不断增加，除非页面在此期间被再次引用，导致其使用位为1；
- 把t_off与系统时间参数t_max相比，若t_off>t_max，就把页面从工作集中移出，释放相应页框。

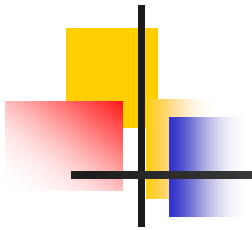
4、缺页频率替换算法

- 缺页频率替换算法（Page Fault Frequency，PFF)
  - 根据连续的缺页之间的时间间隔来对缺页频率进行测量，每次缺页时，利用测量时间调整进程工作集尺寸。
- 规则：如果本次缺页与前次缺页之间的时间超过临界值t，那么，所有在这个时间间隔内没有引用的页面都被移出工作集。

PFF例(t=2，>2触发换出检查)

时间	0	1	2	3	4	5	6	7	8	9	10
引用序列		3	3	4	2	3	5	3	5	1	4
1	√	√	√	√						√	√
2					√	√	√	√	√		
3		√	√	√	√	√	√	√	√	√	√
4	√	√	√	√	√	√	√	√	√		√
5	√	√	√	√			√	√	√	√	√
换入		3			2		5			1	4
换出					1，5					2，4	
工作集合		1，3，4，5	1，3，4，5	1，3，4，5	2，3，4	2，3，4	2，3，4，5	2，3，4，5	2，3，4，5	1，3，5	1，3，4，5

缺页数：5，缺页率50%



- 另一种PFF算法实现：
  - 设置两个阈值，当缺页率达到上限值时为进程增加物理块，当缺页率达到下限值时减少进程的物理块
- PFF算法的缺点：
  - 当进程由一个局部转移到另一个局部时，在原局部中的页面未移出内存之前，连续的缺页会导致该进程在内存的页面迅速增加，产生对内存请求的高峰。

7.8 页面大小的选择

- 页面大小的选择涉及诸多因素，需要在这些因素之间权衡利弊。
  - 页表大小：页面越小，页表越长。依此应选择大页面。
  - 页内碎片问题：页面越大，碎片越大。依此应选择小页面。
  - 内外存传输：大小页面传输时间基本相同，依此应选择大页面。
  - 页面大小与主存关系：主存大则页面大。
  - 程序结构对缺页率的影响：好的程序结构可以降低缺页率。
  - 页面大小与快表命中率的关系：页面小快表命中率低。

页面大小选择的分析

- 设系统内每个进程的平均长度为s，页面大小为p，每个页表项需e个字节。
  - 每个进程占用页面数量：s/p
  - 页表占用空间：se/p
  - 每进程碎片的平均大小：p/2
  - 每进程浪费的空间为：se/p+p/2
  - 对p求导数：$=1/2-se/p^2$
  - 令导数为0求最小值：$1/2s-e/p^2=0$
  - 则页面大小为：$p=\sqrt{2es}$时，浪费最小。

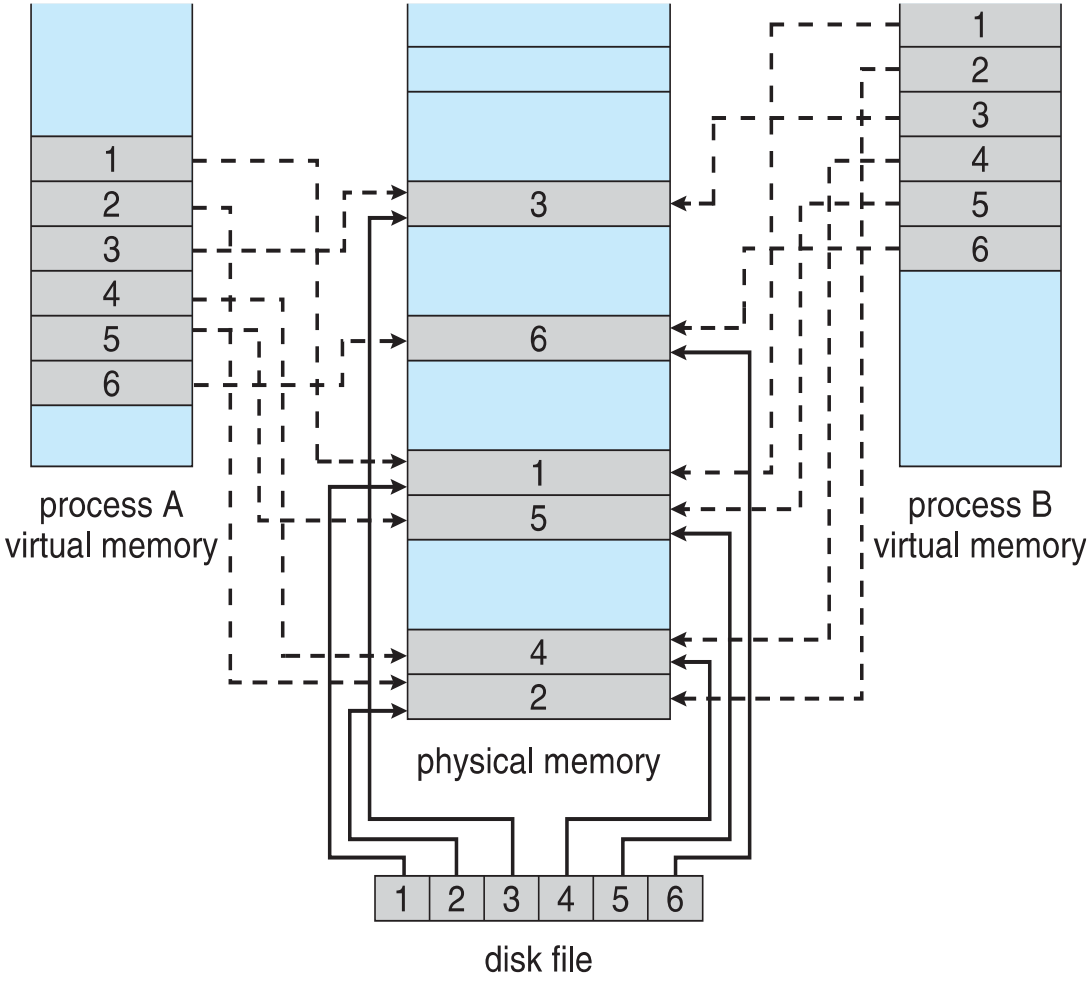
7.9 内存映射文件

- 磁盘文件的访问方式
  - 标准I/O方式：read(), write(), open()…
    - 系统调用&磁盘访问
  - 内存映射文件机制
    - 将文件块映射到内存页中，允许按照内存的方式来访问文件块。
- 基本技术思路
  - 文件按照调页管理方式进行访问
  - 一个页面大小的文件会被读入物理内存页
  - 接下来的访问被作为普通内存访问进行处理
- 可通过此机制实现进程对相同文件的共享
- 如何处理写数据?
  - 定期处理，或者在close()时处理
  - 扫描到脏页时处理

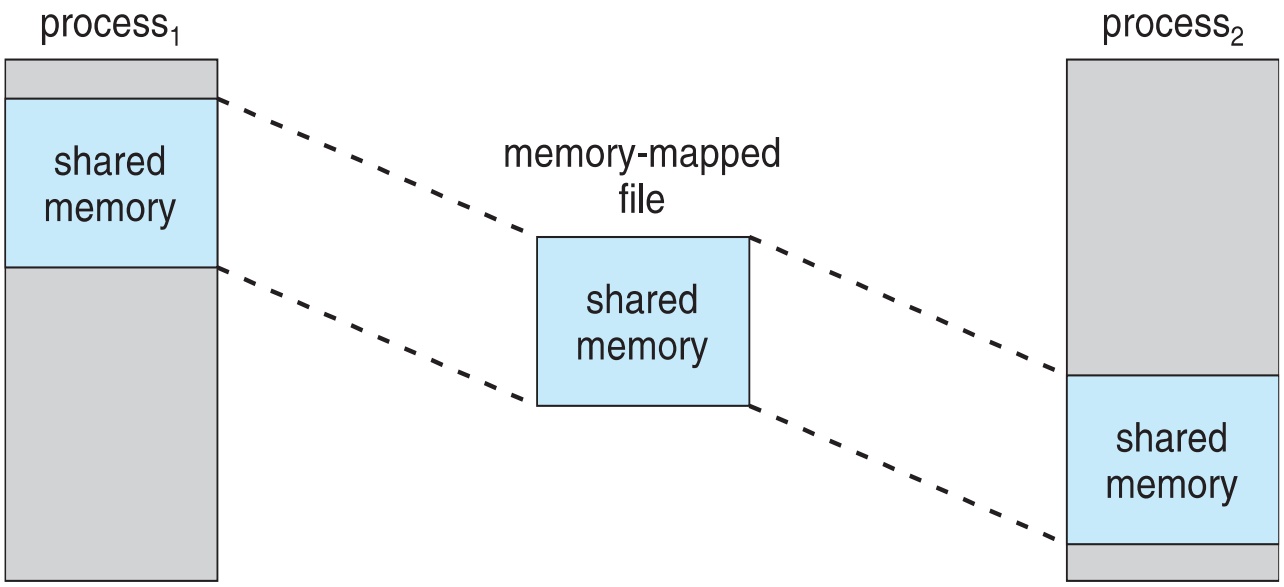
内存映射文件用于I/O

- OS使用内存映射直接替代标准的I/O
- 对于显示访问内存映射文件：
  - Linux：mmap()
  - Win32: CreateFileMapping, MapViewOfFile()
- 对于标准I/O：read()，write()
  - 映射文件到内核空间
  - I/O数据先传输到内核空间，然后在传输到用户空间
- COW可用于处理只读的共享页面，以及被修改后的复本。

内存映射文件示例图

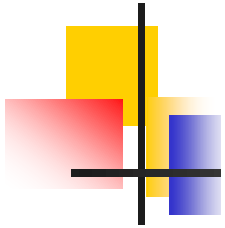

1
2
3
4
5
6
process A
virtual memory
3
6
1
5
4
2
physical memory
1
2
3
4
5
6
process B
virtual memory
1
2
3
4
5
6
disk file

通过内存映射，实现共享内存



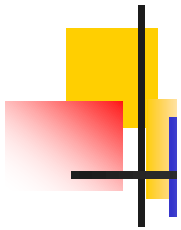
Windows内存共享例子

- 基本思想
  - 创建**文件映射(file mapping)**，建立映射文件在各进程虚拟地址空间的**视图(View)**
- 以生产者/消费者为例
  - 生产者使用内存映射特性创建内存共享对象
  - 使用`CreateFile()`创建一个文件， 返回一个`HANDLE`
  - 通过 `CreateFileMapping()` 创建映射，得到一个 **named shared-memory object**
  - 通过`MapViewOfFile()`创建虚拟地址视图
- 例子如下：



```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hFile, hMapFile;
   LPVOID lpMapAddress;

   hFile = CreateFile("temp.txt", /* file name */
      GENERIC_READ | GENERIC_WRITE, /* read/write access */
      0, /* no sharing of the file */
      NULL, /* default security */
      OPEN_ALWAYS, /* open new or existing file */
      FILE_ATTRIBUTE_NORMAL, /* routine file attributes */
      NULL); /* no file template */

   hMapFile = CreateFileMapping(hFile, /* file handle */
      NULL, /* default security */
      PAGE_READWRITE, /* read/write access to mapped pages */
      0, /* map entire file */
      0,
      TEXT("SharedObject")); /* named shared memory object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* write to shared memory */
   sprintf(lpMapAddress,"Shared memory message");

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hFile);
   CloseHandle(hMapFile);
}
```

Figure 9.24 Producer writing to shared memory using the Windows API.



```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hMapFile;
   LPVOID lpMapAddress;

   hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W access */
      FALSE, /* no inheritance */
      TEXT("SharedObject")); /* name of mapped file object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* read from shared memory */
   printf("Read message %s", lpMapAddress);

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hMapFile);
}
```

Figure 9.25 Consumer reading from shared memory using the Windows API.

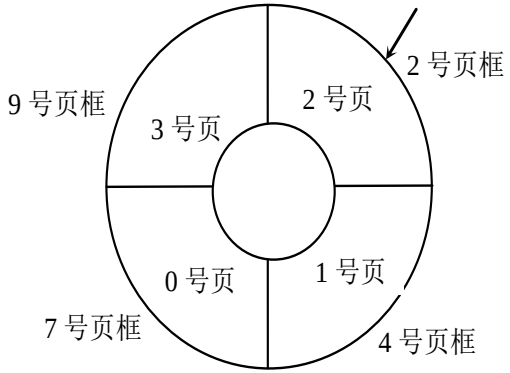
内存映射I/O

- 对于I/O控制器都具有命令寄存器、数据寄存器等
- 内存映射I/O：
  - 用一组内存地址专门映射到设备寄存器
  - 对内存地址的访问方式==对设备寄存器的访问
  - 如：视频控制器的访问，屏幕位置和内存位置一一对应

作业1:

- 设某计算机的逻辑地址空间和物理地址空间均为64KB，按字节编址。若某进程最多需要6页数据存储空间，页的大小为1KB。操作系统采用固定分配局部置换策略为此进程分配4个页框（Page frame），如下表所示。

页号	页框号	装入时刻	访问位
0	7	130	1
1	4	230	1
2	2	200	1
3	9	160	1



当该进程执行到260时刻时，要访问逻辑地址为17CAH的数据，请回答下列问题：

（1）该逻辑地址对应的页号是多少？

（2）若采用先进先出（FIFO）置换算法，该逻辑地址对应的物理地址是多少？要求给出计算过程。

（3）若采用时钟（CLOCK）置换算法，该逻辑地址对应的物理地址是多少？要求给出计算过程。（设搜索下一页的指针沿顺时针方向移动，且当前指向2号页框，如图所示）。

作业2

- 请求分页管理系统中，假设某进程的页表内容如下表所示：

页号	页框号	有效位
0	101H	1
1	-	0
2	254H	1

页面大小为4KB，一次内存的访问时间是100ns，一次快表（TLB）的访问时间是10ns，处理一次缺页的平均时间是10^8ns（已含更新TLB表和页表的时间），进程的驻留集大小固定为2，采用最近最久未使用替换算法（LRU）和局部淘汰策略。假设：

（1）TLB初始为空

（2）地址转换时先访问TLB，若TLB未命中，再访问页表（忽略访问页表之后的TLB更新时间）

（3）有效位为0表示页面不在内存，产生缺页中断，缺页中断处理后，返回到产生缺页中断的指令处重新执行。设有虚地址访问序列2362H，1565H，25A5H，请问：

（1）依次访问上述三个虚地址，各需多少时间？给出计算过程。

（2）基于上述访问序列，虚地址1565H的物理地址是多少？请说明理由

作业3

- 某请求分页系统的页面置换策略如下：从0时刻开始扫描，每隔5个时间单位扫描一轮驻留集（扫描时间忽略不计）且在本轮没有被访问过的页框将被系统回收，并放入到空闲页框链尾，其中内容暂时不清空，当发生缺页时，如果该页曾被使用过且还在空闲页框链表中，则将其重新放回进程的驻留集中；否则，从空闲页框链表头部取出一个页框。
- 忽略其他进程的影响和系统开销。初始时进程驻留集为空。目前系统空闲页的页框号依次为32、15、21、41。进程P依次访问的<虚拟页号,访问时刻>为<1,1>、<3,2>、<0,4>、<0,6>、<1,11>、<0,13>、<2,14>。请回答以下问题：

(1) 当虚拟页为<0,4>时，对应的页框号是什么?

(2) 当虚拟页为<1,11>时，对应的页框号是什么？说明理由。

(3) 当虚拟页为<2,14>时，对应的页框号是什么？说明理由。

(4) 这种方法是否适合于时间局部性好的程序？说明理由。

作业4

- 使用OPT、FIFO、LRU、简单时钟四种方法，比较如下执行序列的性能。假设采用固定分配策略，进程可分页框总数为3个，执行中所引用到的页面总数为5个，执行序列为:

  2 3 2 1 5 2 4 5 3 2 5 2

作业5

- P301 9.9