

虚拟内存

阅读任务

- 第23节 “The VAX/VMS Virtual Memory System”

本章教学目标

- 理解虚拟内存的概念
- 理解缺页中断和请求分页机制
- 掌握请求分页系统的虚实地址转换过程
- 掌握页面替换算法
- 掌握工作集的概念及基于工作集的页面替换方法

大纲

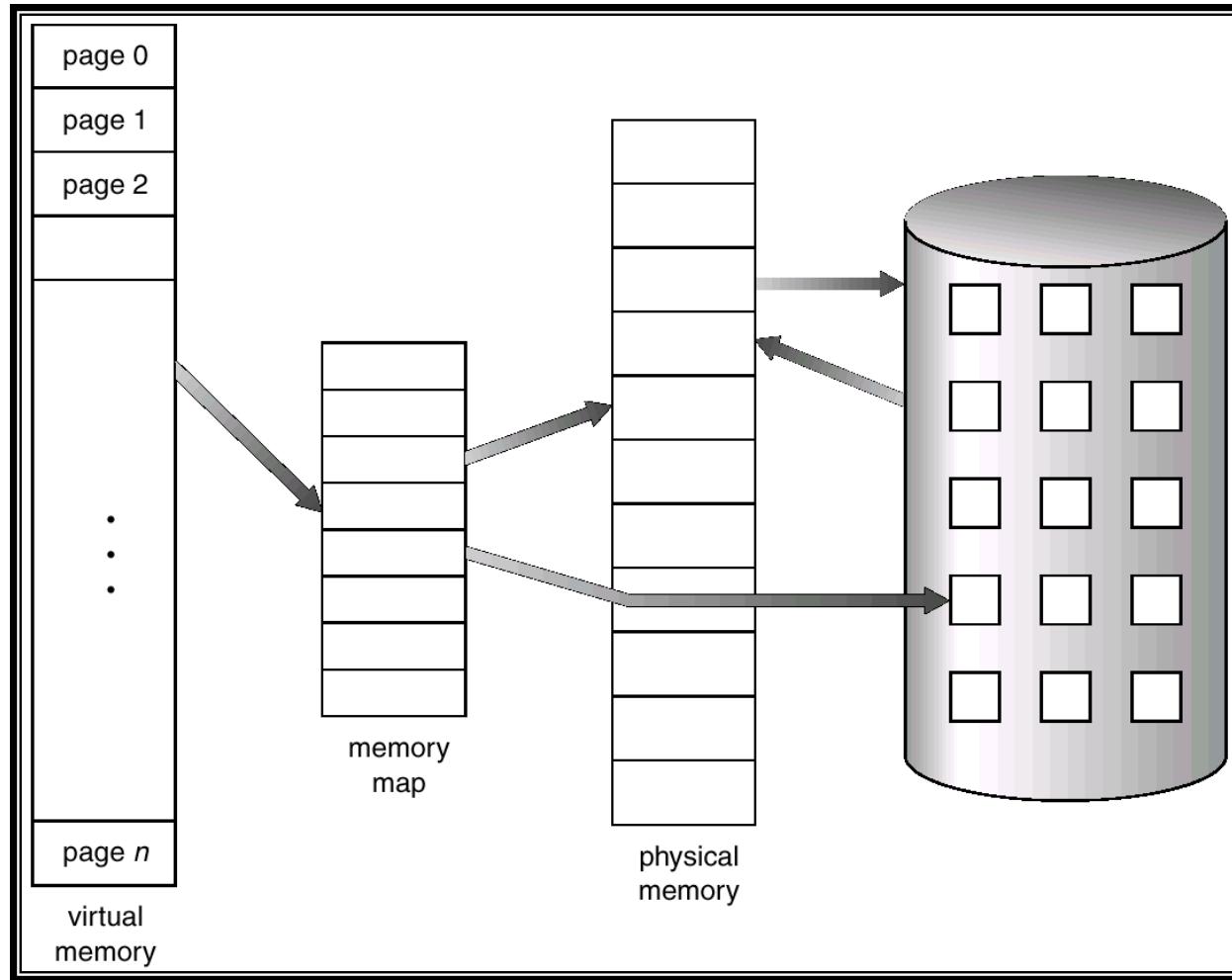
- 虚拟内存的概念
- 请求分页虚拟存储管理
- 请求分段虚拟存储管理
- 请求段页式虚拟存储管理

虚拟内存的概念

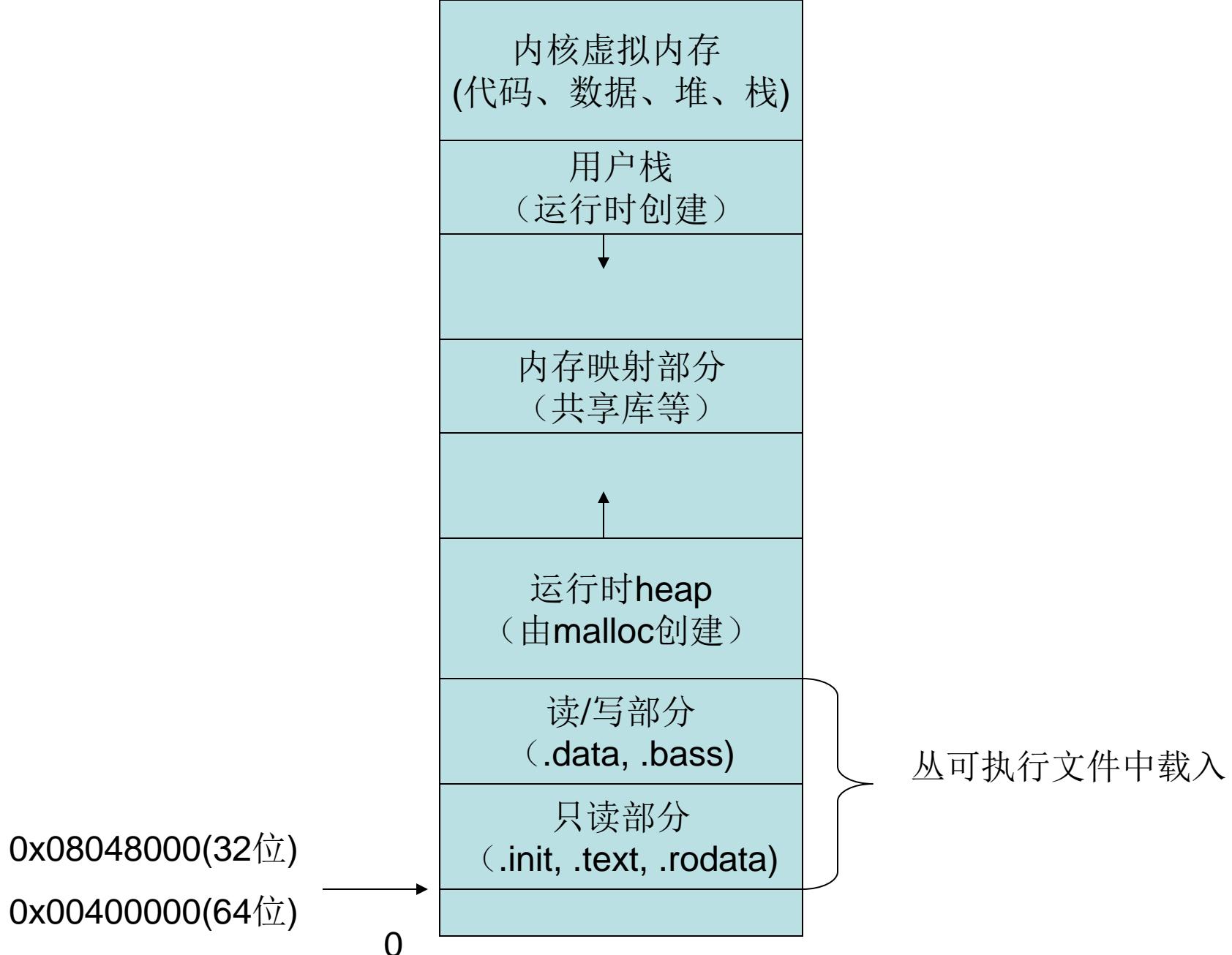
- 实存管理
 - 必须为进程分配足够多的主存空间，装入其全部信息，否则进程无法运行
 - 程序的大小受物理内存的限制
 - 装入全部程序是不必的，有些代码很少被使用，如异常处理代码
 - 数组，表等通常分配比需要的更大的空间
- 虚拟内存管理
 - 将用户的逻辑内存与物理内存分开
 - 程序部分装入内存即可运行(分页、分段使得程序可以离散载入内存)
 - 逻辑地址空间可以比物理地址空间大
 - 可以容纳更多的进程，提高并发度
 - 进程对换时无需对换整个进程，减少了I/O操作，提高了每个进程的运行速度
 - 虚拟内存一般都基于分页存储管理

虚拟内存的概念

- 定义：
 - 在具有层次结构存储器的计算机系统中，自动实现部分装入和部分替换功能，能从逻辑上为用户提供一个比物理主存大得多的、可寻址的“主存储器”。
- 优点
 - 用户虚拟地址空间的容量不再受物理主存大小限制，而受限于计算机的地址结构和可用的磁盘容量
 - 单个进程的需求大于主存空间时，进程可以运行
 - 多个进程的存储需求总量超出主存容量时，也可以都装入内存，实现多道程序运行



用户编程的逻辑地址空间比物理内存大



虚拟地址空间虚拟页面的分类

- 未分配
 - 没有任何数据与该虚拟页关联
 - 不占用磁盘空间
- 分配且在内存
 - 在物理内存缓存了
- 分配但不在内存
 - 还没载入内存

注：通常一个进程只会使用其虚拟地址空间中的一小部分，大部分都是未分配

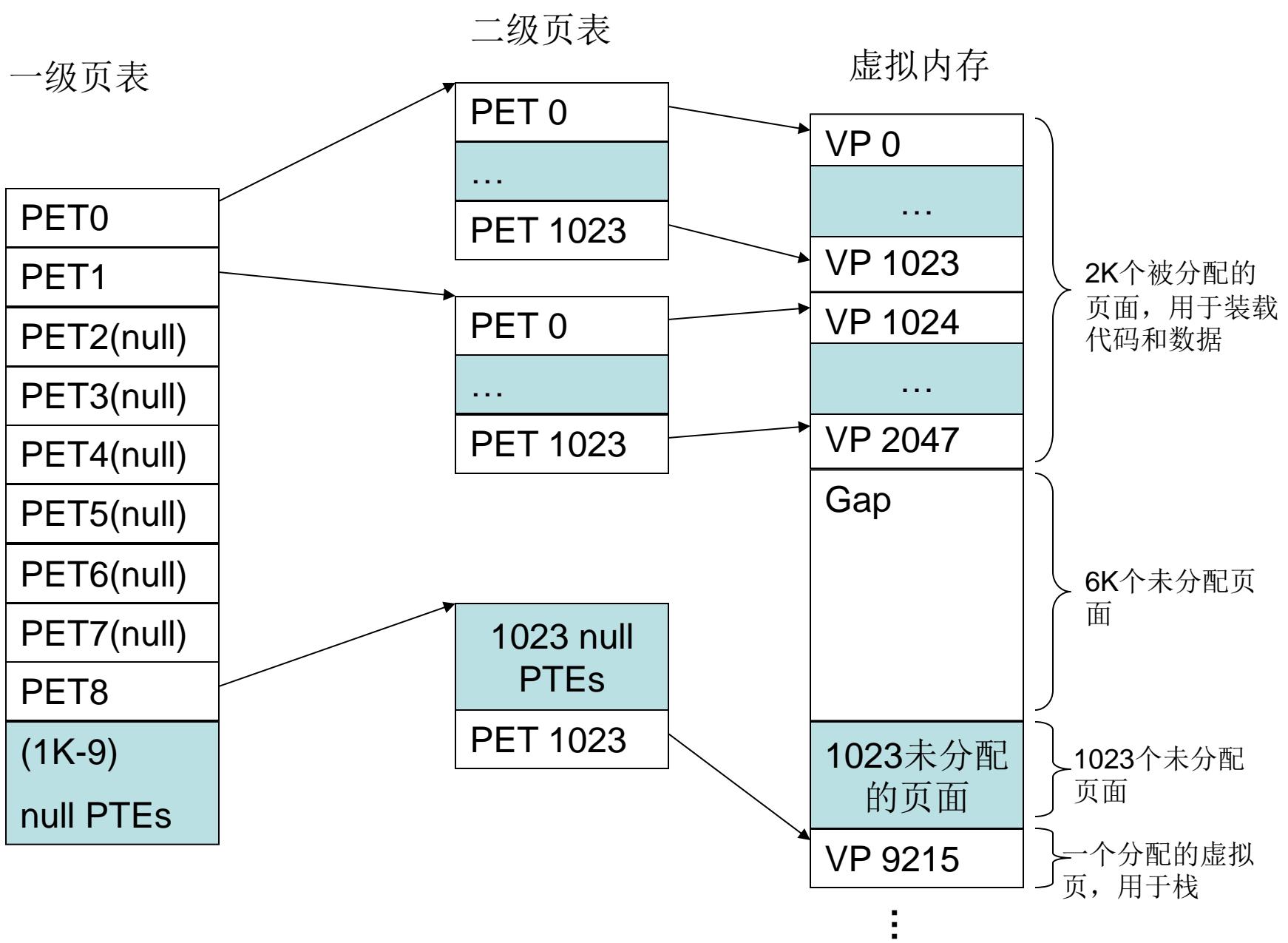
二级页表与虚拟内存

- 一级页表中页表需要全部载入内存且连续存放
 - 页表空间大
 - 连续空间难以找到
- 但是：
 - 应用程序通常只使用整个虚拟地址空间的一小部分
 - 无需为所有虚拟地址空间都分配页表项
 - 即便是应用程序使用的虚拟地址空间，程序的局部性原理导致在某个时间段只有一部分页会被访问，有些页面在整个程序生命周期都不会被访问
 - 并非所有页表项都需要常驻内存
- 解决方法：二级页表

二级页表与虚拟内存

- 假设：
 - 32位虚拟地址
 - 4KB的页面
 - 页表项4字节
 - 虚拟地址空间的组成
 - 前2K个虚拟页面被分配存放代码和数据
 - 之后6K个虚拟页面未分配
 - 之后1023个页面也未分配
 - 之后一个页面用于用户栈

如果采用一级页表，则每个进程的页表本身需要
 $4 * 2^{20} = 4\text{MB}$, what about 64位虚拟地址空间?



仅需载入一级页表(4KB)和三个二级页表, 总共16KB
 二级页表和虚拟内存结合, 可以大大降低内存中页表的存储空间

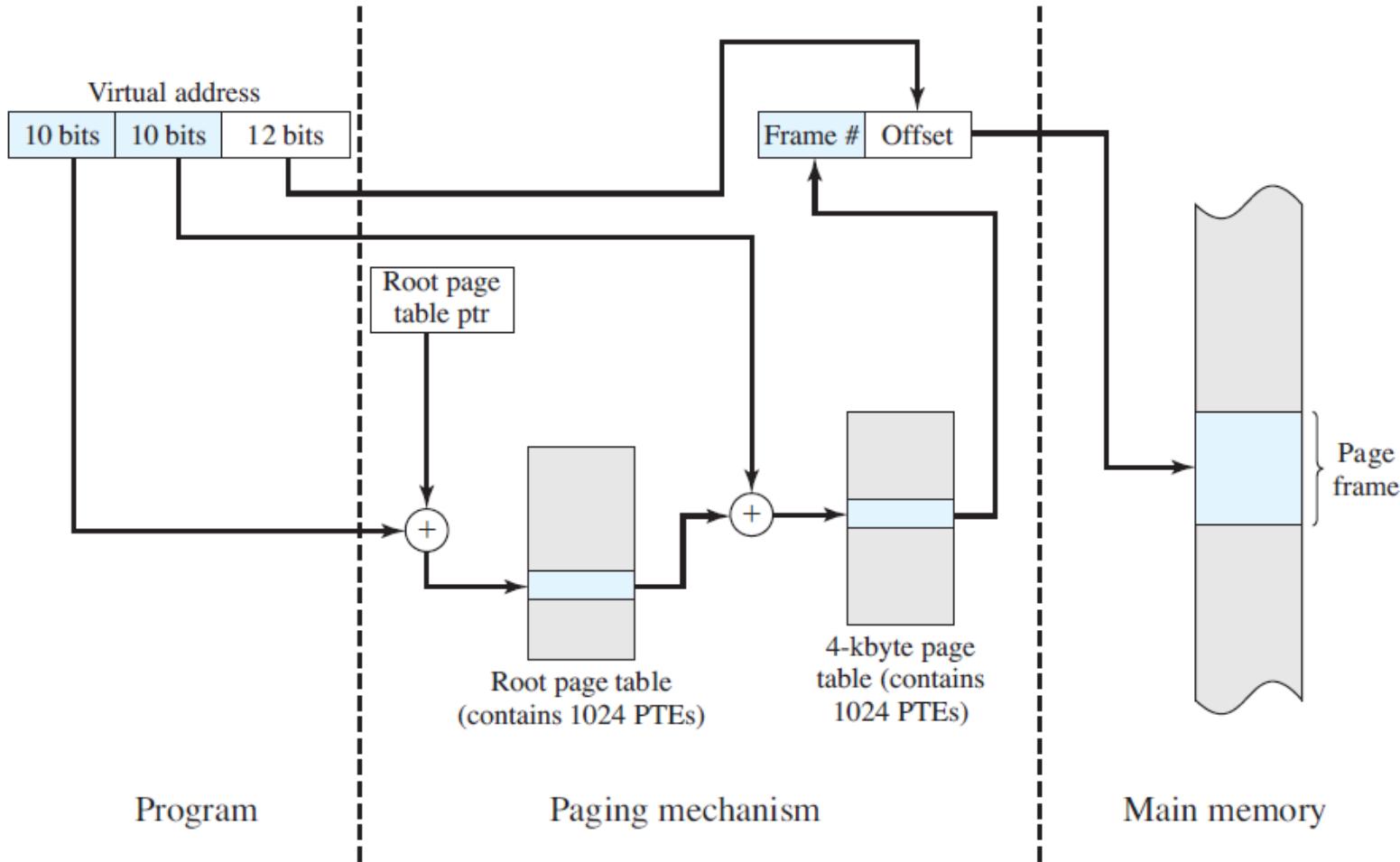


Figure 8.5 Address Translation in a Two-Level Paging System

- 一级页表驻留内存
- 二级页表可以按需载入内存

程序执行的局部性原理

- 程序和数据的访问都有聚集成群的倾向
 - 空间局部性
 - 时间局部性
- 进程运行时无需将全部信息调入内存

虚拟内存与对换方式的比较

- 两者都是在主存和磁盘之间交换信息
- 交换粒度不同
 - 对换以进程为单位，进程要么完全在内存中，要么完全对换到磁盘上
 - 虚拟内存以页或段为单位，进程可以部分位于内存中，部分位于磁盘上

大纲

- 虚拟内存的概念
- 请求分页虚拟存储管理
 - 缺页中断
 - 请求分页地址转换过程
 - 页面分配策略
 - 页面替换算法
 - 工作集理论
- 请求分段虚拟存储管理
- 请求段页式虚拟存储管理

虚拟内存需要解决的问题

- 什么时候将页面或分段取至内存？（装入策略）
 - 请求分页/分段
 - 预调
- 为每个进程分配多少空间？
 - 固定大小？
 - 可变大小？
- 进程放在内存中的什么地方？
 - 分页存储管理：无所谓
 - 分段存储管理：采用连续存储空间管理中的分配方法
- 页面替换算法
 - 当新页面要载入内存，但没有空间时，需要选择一个替换出内存，选择哪个？
- 页面替换范围
 - 全局？
 - 局部？

页面装入策略

• ~~①~~ 请页式(demand paging)

- 仅当需要访问程序和数据时，通过缺页中断并由缺页中断处理分配页框，并将所需页面装入主存
- 只有**被访问的页面才会被调入主存**，节省内存空间。
- 处理缺页中断的次数多，I/O操作次数多
- 现代操作系统基本都用的是请页式

• ~~②~~ 预调式

- 操作系统根据某种算法，预测进程最可能要访问的那些页面，在使用页面前预先调入主存。
- 进程的页面大多连续存放在磁盘，一次I/O操作可以调入多个页面。
- **有可能调入不被访问的页面**

请求分页

- 按需将页面载入内存
 - 更少的I/O
 - 更小的内存需求
 - 更快的响应速度
 - 更多的用户
- 页面的需求 \Rightarrow 访问该页面
 - 非法访问 \Rightarrow abort
 - 不在内存 \Rightarrow 载入内存
- 需要能够区分在内存中的分页与不在内存中的分页

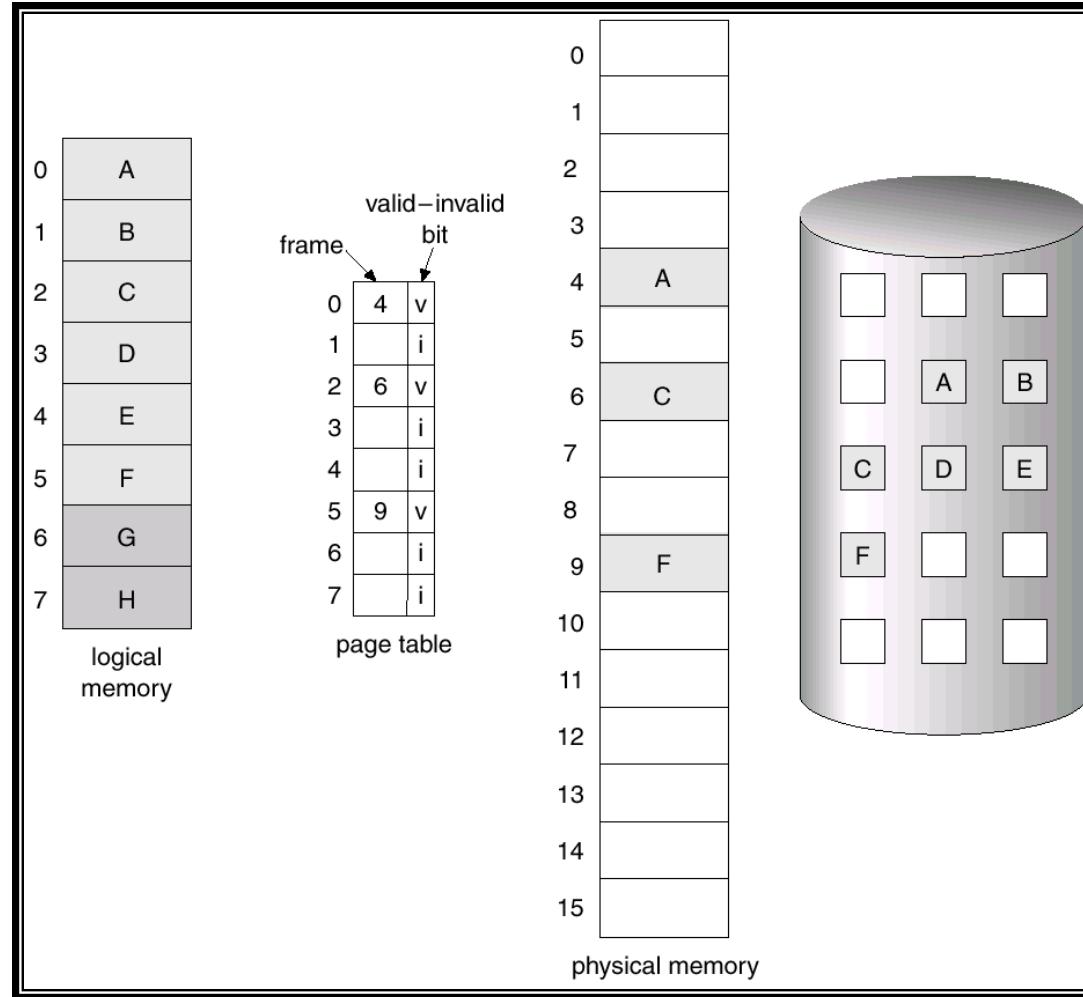
Valid-Invalid位

- 每个页表项都包含一个valid-invalid 位， ($1 \Rightarrow$ in-memory, $0 \Rightarrow$ not-in-memory)
- 起初valid-invalid位被置为0，一旦相应的页被载入内存，则被置为1.
-

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

page table

页表的例子



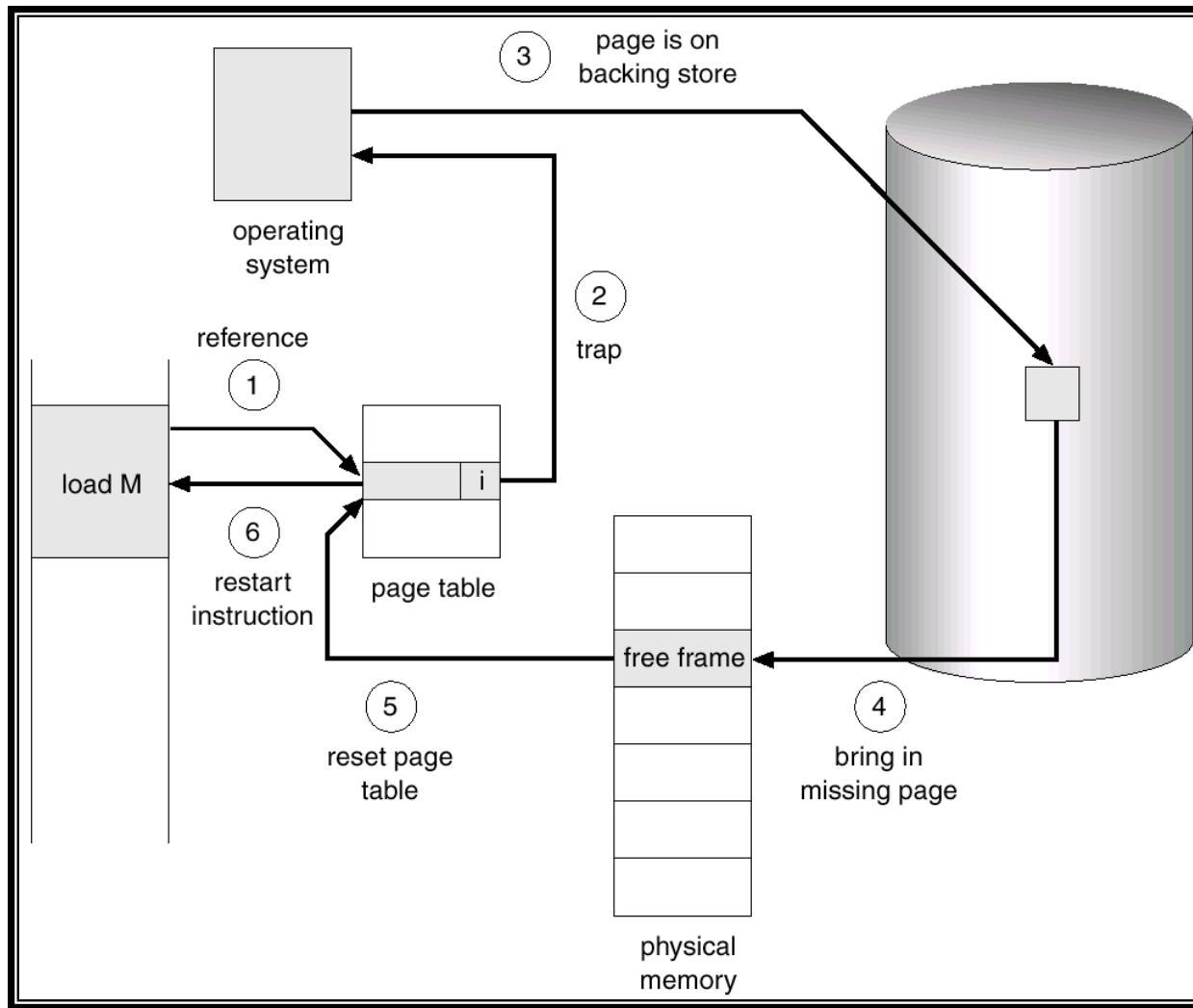
Valid-invalid 位为1的表项给出了页面在物理内存中的frame号；而valid-invalid位为0的表项，表项中的内容可以为空，也可以包含页面在磁盘上的地址。

- 页表中的页面不在内存中的后果
 - 若进程的生命周期都不访问该页面，则无影响
 - 若进程在执行过程中访问不在内存中的页面，则产生**缺页中断/异常**，从而陷入操作系统，执行相应的**缺页中断处理程序**

缺页异常/缺页中断

- 如果进程访问了不在内存中的页面，则将产生缺页异常，陷入操作系统，由操作系统处理缺页异常
- 缺页中断在指令执行期间产生并获得处理 *否则指令无法执行完*
 - 与一般的中断响应时机不同
 - 一条指令执行期间可能产生多次缺页中断
- 操作系统对缺页异常的处理:
 - 若为非法地址访问，则终止程序；
 - 若是合法地址访问，但页面不在内存，则：
 - 获得空闲页框
 - 若不存在空闲页框，则需进行页面替换
 - 启动I/O操作，将页面从磁盘载入内存中的空闲页框；
 - 修改页表中的对应表项，填入页框号,将 valid-invalid位置1.
 - 重启由于缺页异常而被中断的指令

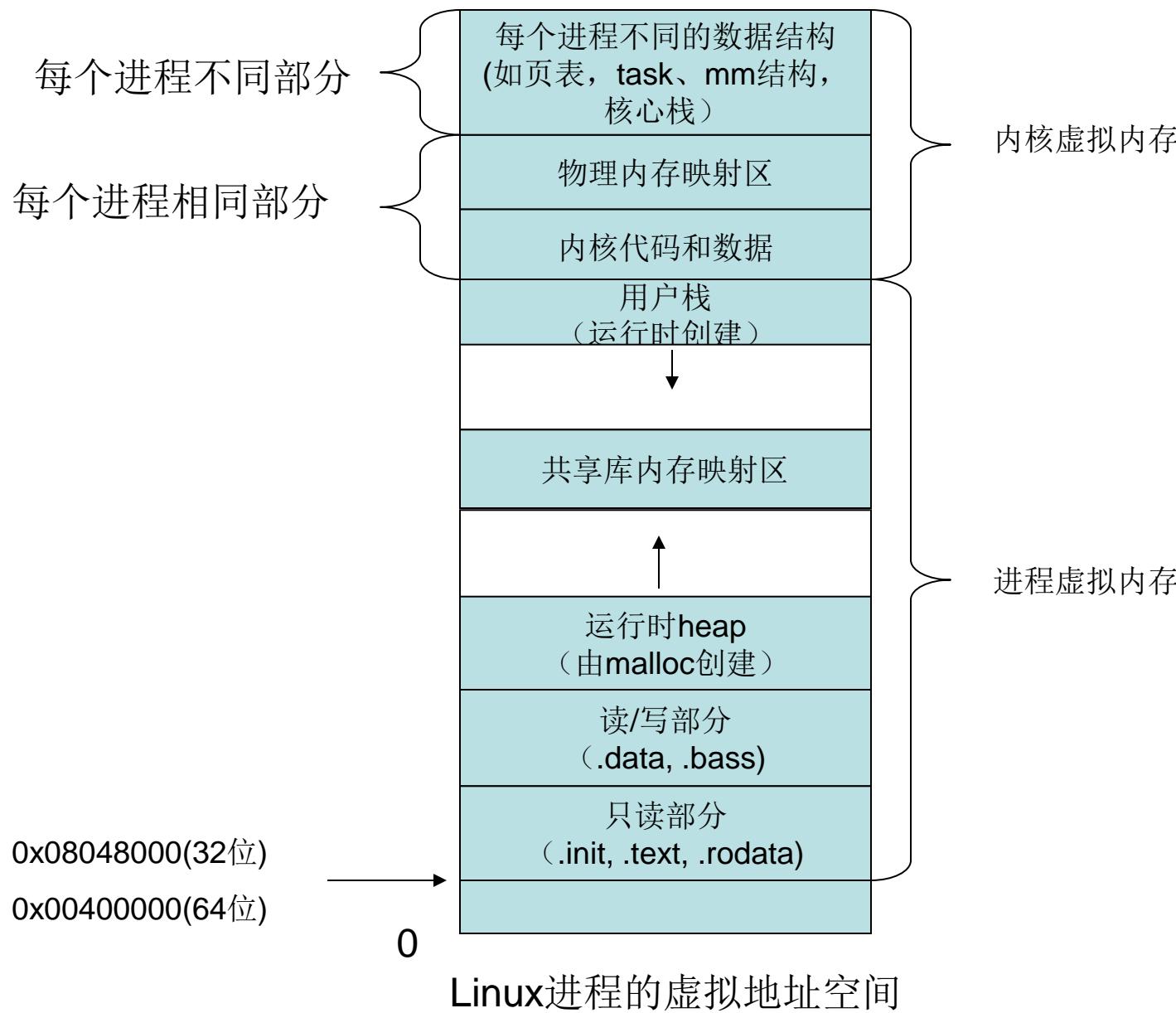
缺页异常处理过程



缺页查找

- 操作系统可以为每个进程维护一个外页表
- 进程启动运行前系统为进程建立外页表，
并将进程程序页面装入辅助存储器
- 外页表存放逻辑地址空间的页号和辅助存
储器位置的映射

Linux中的缺页异常处理

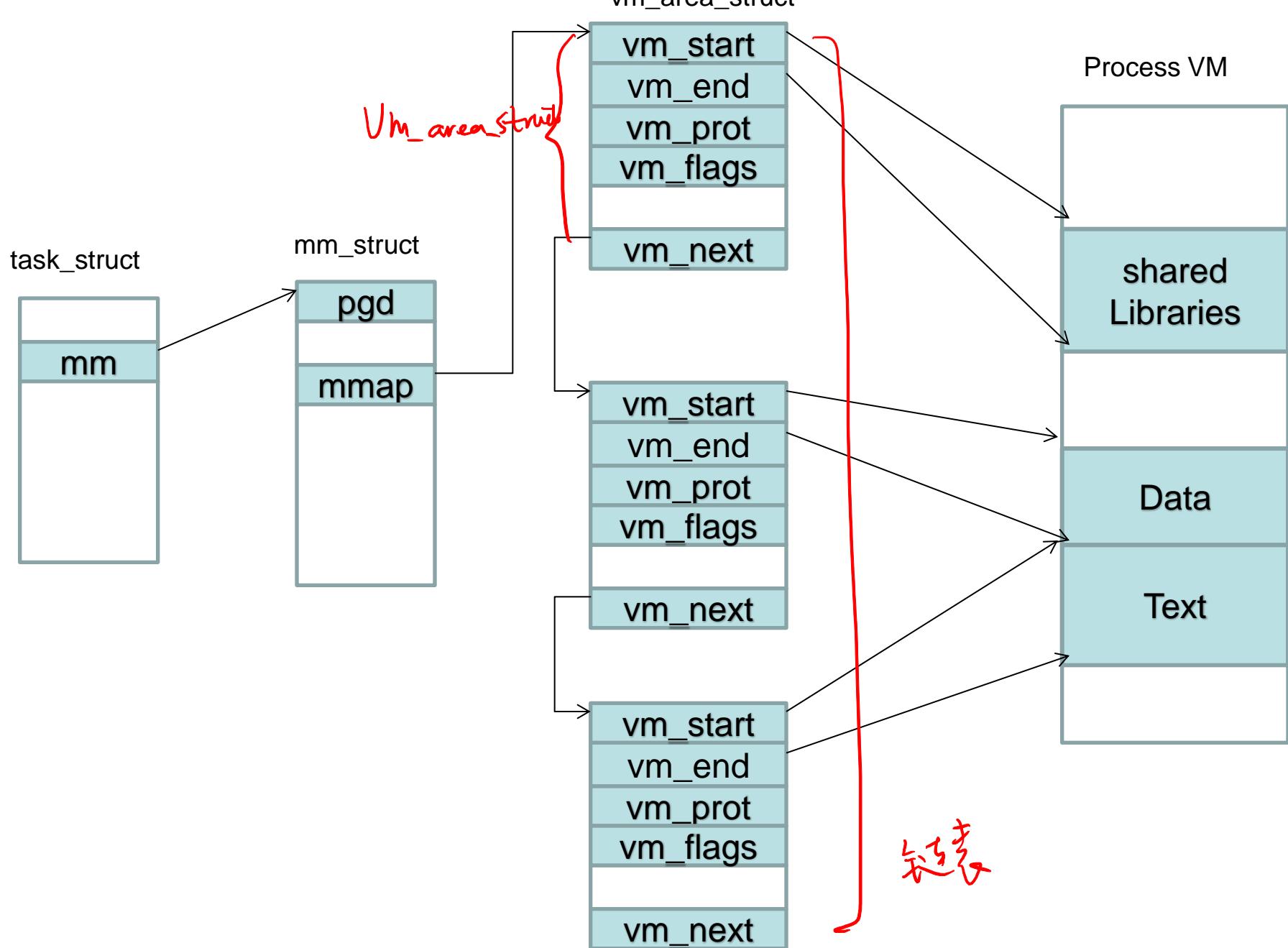


Linux的缺页异常处理

- Linux将虚拟存储组织成一组**segment**集合，每个**segment**都是一组相关的连续已分配页面集合
 - 代码段
 - 数据段
 - 堆段
 - 共享库段
 - 用户栈段
 - ...
- 每个已分配的页面都属于某个**segment**, 不属于任何 **segment**的虚拟页不能被访问
- 从而允许虚拟地址空间有gap
- 内核不跟踪未分配的虚拟页面，未分配页面在内存、磁盘和内核中都不占用额外资源

Linux缺页异常处理

- kernel为每个进程维护一个~~task_struct~~结构（对应进程控制块概念），里面存放了内核为运行该进程所需的所有信息
- ~~task_struct~~中的~~mm_struct~~结构记录了进程当前的虚拟内存状态，其中的~~mmap~~指向了~~vm_area_structs~~(段结构)的链表，每个都刻画了当前虚拟地址空间中一个~~segment~~
- ~~vm_area_structs~~的主要成员
 - ~~vm_start~~: 段开始地址
 - ~~vm_end~~: 段结束地址
 - ~~vm_prot~~: 该段里所有页面的读/写权限 *protection*
 - ~~vm_flags~~: 该段的页面是共享的还是独有的
 - ~~vm_next~~: 下一个~~vm_area_struct~~的指针



Linux缺页中断处理

- 虚拟地址A是否合法?
 - A是否在某个segment定义的start和end之间
 - 需要搜索整个vm_area_struct链表
 - 若不合法，产生segment fault
- 访问权限是否合法?
 - 与vm_prot比较确定
 - 若不合法，产生protection exception
- 正常的缺页中断处理
 - 选择待替换页面予以替换
 - 载入新页面
 - 更新页表

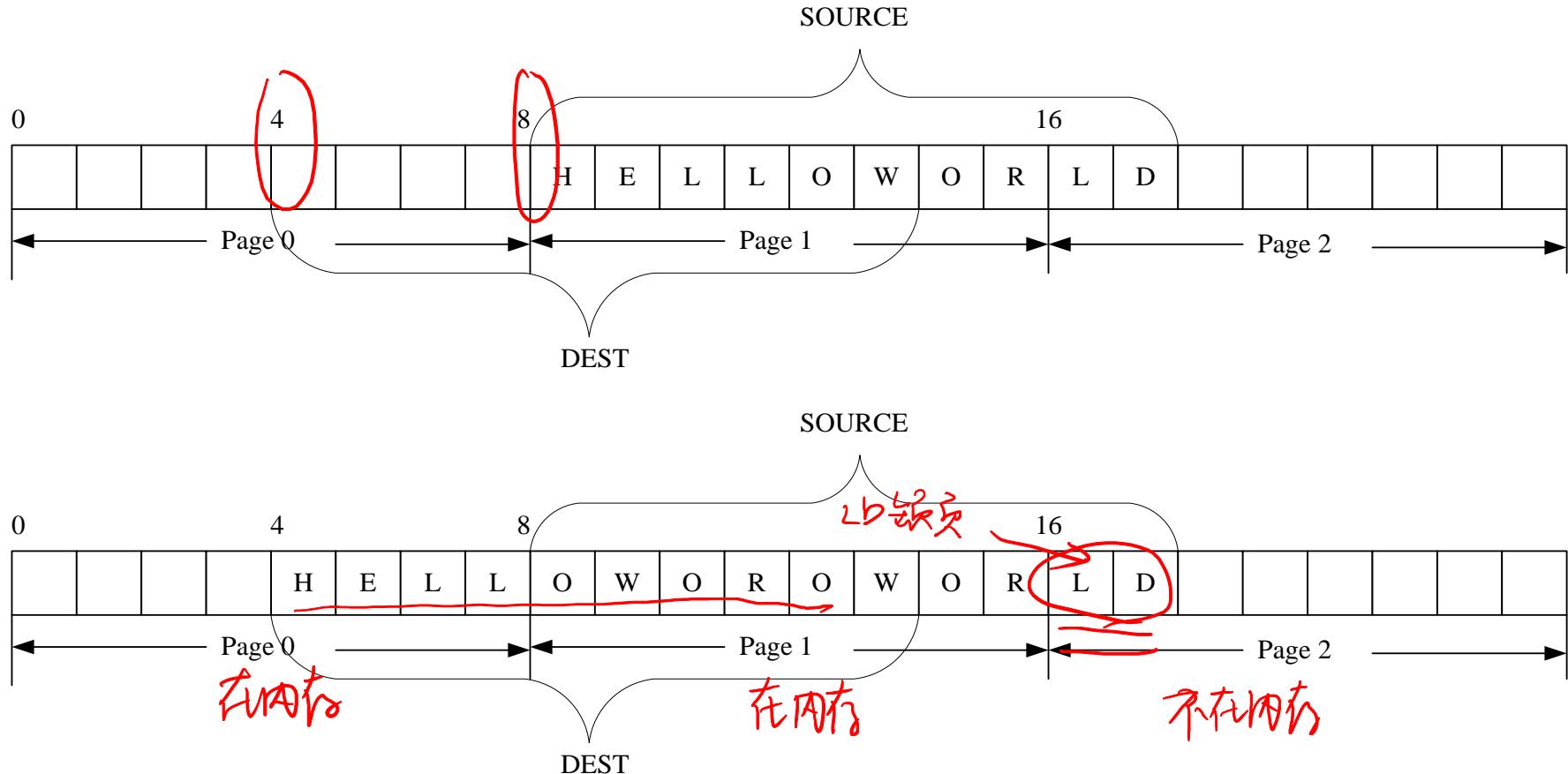
页面替换

- 若需要载入页面时，内存中不存在空闲页框，则需将某些页面对换出内存；
 - 选择哪个页面对换出内存？**页面替换算法！**
 - 页面替换算法的目标是最小化缺页异常的次数

局部替换
< 针对

指令重启

- 一条指令可能访问多个内存页面，从而产生多次缺页中断
- 一般来说，重启指令只需保存处理器现场即可。
- ADD A B C: 将A, B的内容相加，结果存储到C
 - 获取指令并译码操作码ADD
 - 获取A
 - 获取B
 - 执行A+B ④
4次访间内存
 - 将结果存储到C
- 若在存储C时产生缺页中断，则需要重新获取指令，译码，获取操作数，执行相加。
- 当一条指令可能修改多个内存空间时，指令重启不再简单。
 - 例如IBM 360的MVC指令，可以将256个字节从一个地方移动到另一个地方（可能是重叠的），如果指令执行一部分后产生缺页中断，则由于源可能被部分修改，很难简单地重启指令。



假设页大小为8个字节，当指令执行到移动LD时，发生缺页中断，但此时SOURCE部分的HELL已被OWOR改写，导致指令无法重启。

解决方法：

(1) 微指令开始时先访问两个块的起始和终止位置

(2) 将重叠位置的内容写入临时寄存器，并在缺页异常发生前写回内存。收至原样

请求分页的性能

miss.

- 缺页率 $0 \leq p \leq 1.0$
 - $p = 0$, 表明不存在缺页率
 - $p = 1$, 每次访问都产生缺页异常
- 有效访问时间 (EAT)

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

^{命中率}

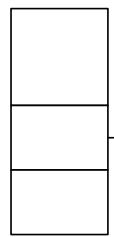
+ page fault overhead 完成开销

+ swap page out [页表替换]

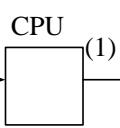
+ swap page in 读入内存

+ restart overhead 重启开销

逻辑空间



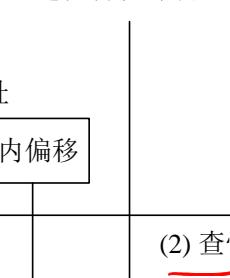
CPU



(1) 地址分解



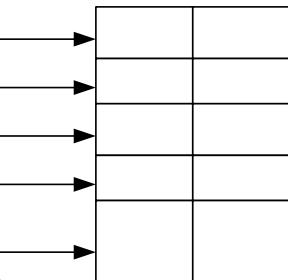
进程切换时装入



运行进程页表基址寄存器
运行进程页表

(2) 查快表

快表



装入快表

(3) 命中

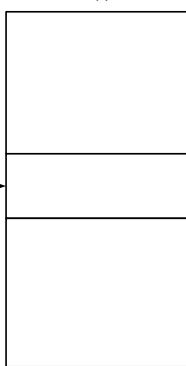
(4) 不命中

物理地址

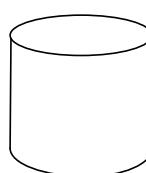


访问

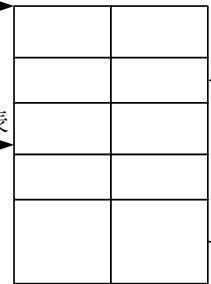
主存



辅助存储器



(8) 装入、改表



(5) 页表命中

内存

(6) 页表不命中

(7) 调页

缺页中断处理

请求分页虚实地址转换过程图

请求分页虚实地址转换过程

- MMU接收CPU传送过来的逻辑地址并自动按页面大小把它分解成两部分：页号和页内位移；
- 以页号为索引搜索快表TLB；
- 如果命中，则返回页框号，并与页内位移拼接成物理地址，然后进行访问权限检查，如果通过，进程就可以访问物理地址；
- 如果不命中，由硬件以页号为索引搜索进程页表，页表基址由硬件页表基址寄存器给出；
- 如果在页表中找到此页面，说明所访问的页面已在主存中，可给出页框号，并与页内位移拼接成物理地址，然后进行访问权限检查，如果通过，则可以访问物理地址，同时将该页面的信息装入快表TLB，以备再次访问；
- 如果页表中的对应页面失效，MMU发出缺页中断，请求操作系统进行处理

页面清除策略

- 请页式
 - 仅当一页被选中进行替换且其被修改过，才把它写回磁盘(修改过才写)
- 预约式
 - 对所有更改过的页面，在需要替换之前把它们都写回磁盘，可成批进行
 - 若在页面真正被替换之前又被修改，则之前的写回操作变的无意义

页面分配

- 为每个进程分配多少初始页面？
- 分配的页面数是否随时间变化而变化？
- 页面分配算法？

页面分配

- 单用户情况下很简单
 - 用户将被分配任何空闲页框
- 问题:请求分页+多道程序设计 平衡?
 - 每个进程得以执行的最小页面数由指令集决定
 - 例如, IBM 370的 MVC(内存到内存的移动)指令,能最多移动256个字符, 需要6个页面得以执行
 - 8 Bytes
 - 指令本身长度为6字节, 可能会跨两个页面
 - 处理from需要两个页面
 - 处理to需要两个页面
 - 两种主要的分配策略:
 - 固定分配
 - 可变分配

$$2+2+2=6 \text{ 大于 } 6$$

页面分配方法

局部性好：分配的页框数少
不好：不均

- 固定分配
 - 进程的生命周期中，保持页框数固定不变
- 可变分配
 - 进程的生命周期中，分配的页框数可以随着进程的执行而变化

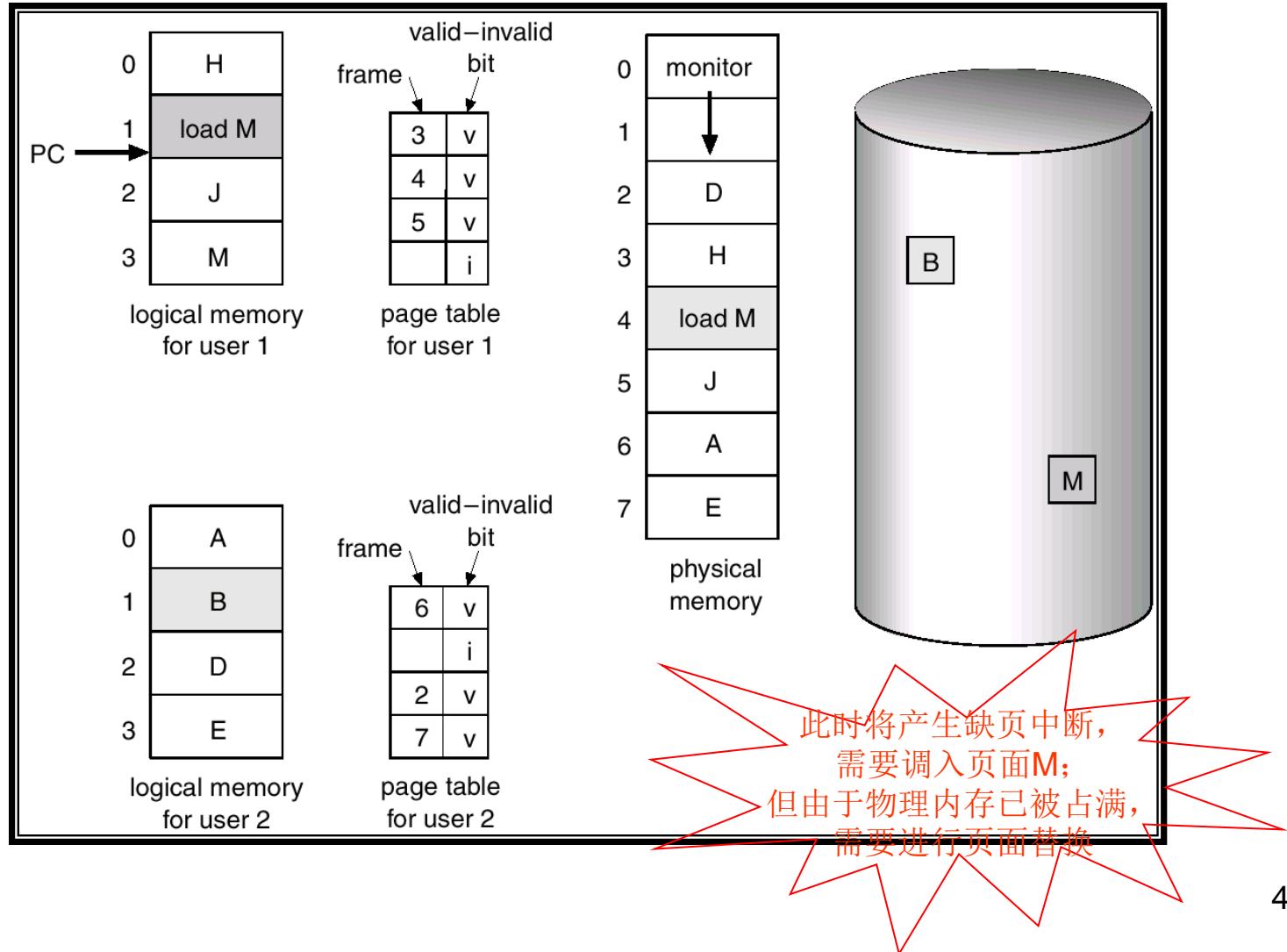
固定分配

- 均匀分配：为每个进程分配相同数量的页面
 - 例如，100个页框，5个进程，则每个进程可以分配20个页面
- 按比例分配
 - 依据进程的大小分配
 - S_j 进程 P_j 的大小
 - $S = \sum S_j$
 - m = 总页框数
 - $a_j = \text{为 } P_j \text{ 分配的页框数} = S_j / S * m$ $m \times \frac{S_j}{S}$
 - If $m = 64$, $S_1 = 10$, $S_2 = 127$ then
 $a1 = 10/137 * 64 \approx 5$
 $a2 = 127/137 * 64 \approx 59$
 - 固定分配一般和局部页面替换算法结合使用

可变分配

- 操作系统保留若干空闲页框
- 当进程发生缺页时，从系统空闲页框中对其分配，把所缺页面调入此页框
- 产生缺页的进程的主存空间会逐渐增大，有助于降低缺页中断总次数
- 可变分配一般可以和全局页面替换算法及局部页面替换算法结合使用

页面替换算法的需求



缺页中断率

- 假设进程在执行过程中访问页面时，页面在内存中的次数为S，而不在内存中的次数为F，则缺页中断率定义为：

$$f = \frac{F}{S+F}$$

$$\frac{F}{S+F}$$

- 影响缺页中断率的因素

- 分配给进程的页框数 $\leftarrow p$

- 页面大小 fixed

- 页面替换算法

- 程序特性 - 局部性高 (数据按行存储)

```
int A[128][128];
```

```
for(int j = 0; j<128; j++)
```

```
    for(int i = 0; i<128; i++)
```

```
        A[i][j]= 0;
```

按列访问

00
10
20

```
int A[128][128];
```

```
for(int i = 0; i<128; i++)
```

```
    for(int j = 0; j<128; j++)
```

```
        A[i][j]= 0;
```

按行访问

00
01
02
03
04
⋮
127
10

若数组按行存放，页面大小为128个字（即正好存放一行数组元素），进程仅分配一个页框。

则左边的程序将产生 128×128 次缺页中断，而右边的仅产生128次中断

页面替换算法

- 目标：最小化缺页中断次数 ↓
- 给定一个页面访问序列，统计给定页面替换算法所产生的缺页中断次数
- 在随后考虑的页面替换算法中，页面访问序列为：

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

全局 vs. 局部替换

- 全局替换
 - 页面替换算法的作用范围是整个系统，不考虑进程属性
 - 一个进程可以从另一个进程获得页框，进程占用的页框数可变
 - 进程可能无法控制自身的缺页率（自己会被别人剥夺）
- 局部替换
 - 页面替换算法的作用范围局限于进程自身
 - 即便其它内存可用，进程也会由于分配给自身的页面不足而执行缓慢
 - 能否动态调整分配给一个进程的页框数？
 - 固定分配 vs 可变分配

全局 vs. 局部替换

页面替换范围 / 页面分配策略	局部替换	全局替换
固定分配	<ul style="list-style-type: none">为每个进程分配的页框数固定被替换的页面必须是<u>属于分配给该进程的页框</u>	不可行
可变分配	<ul style="list-style-type: none">为每个进程分配的页框数会随着时间的变化而变化，以维护<u>工作集</u>； <u>即</u>被替换的页面必须是属于分配给该进程的页框	<ul style="list-style-type: none">被替换的页面可以是内存中所有的页面（<u>不限于本进程</u>）；导致进程的驻留页面集大小会变化

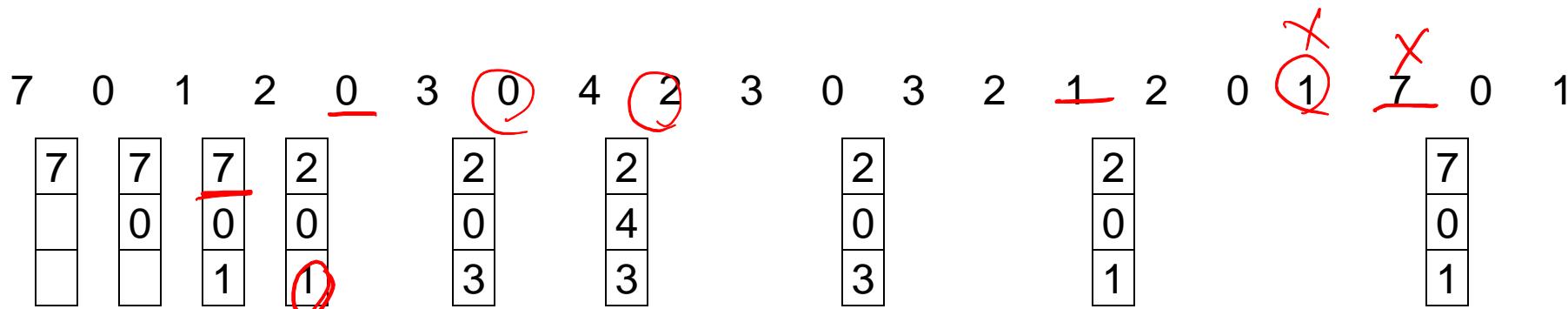
页面替换算法

理论 最佳页面替换算法(OPT) ^{optimum} 知道未来访问序列

- 先进先出页面替换算法(FIFO)
- 最近最少使用页面替换算法(LRU)
- 第二次机会页面替换算法
- 时钟页面替换算法

最佳页面替换算法

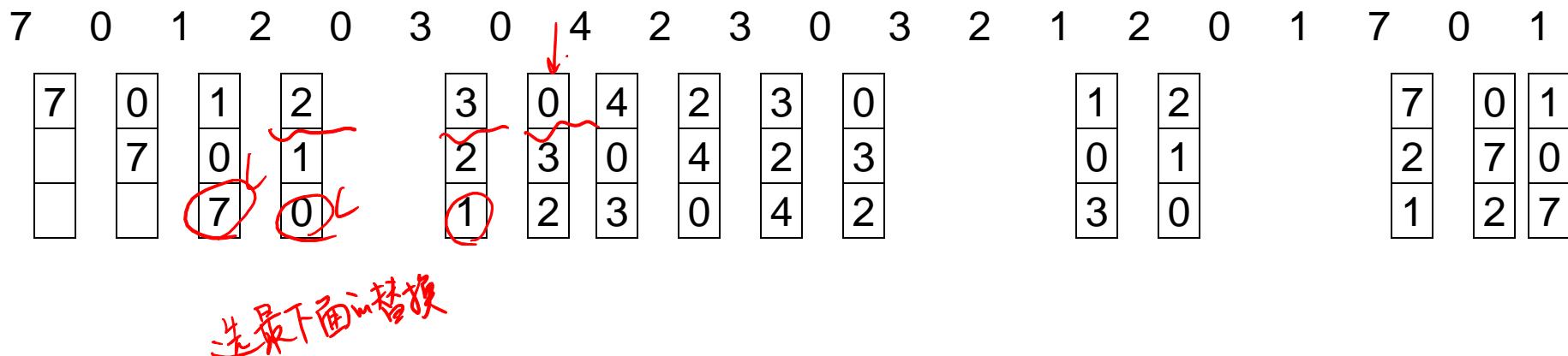
- 淘汰以后不再访问的页，或距离现在最长时间后要访问的页面。
X long time
- 产生最少的缺页数，但是需要预知程序的页面访问序列
- 作为衡量其它页面调度算法优劣的理论最佳算法。



先进先出页面替换

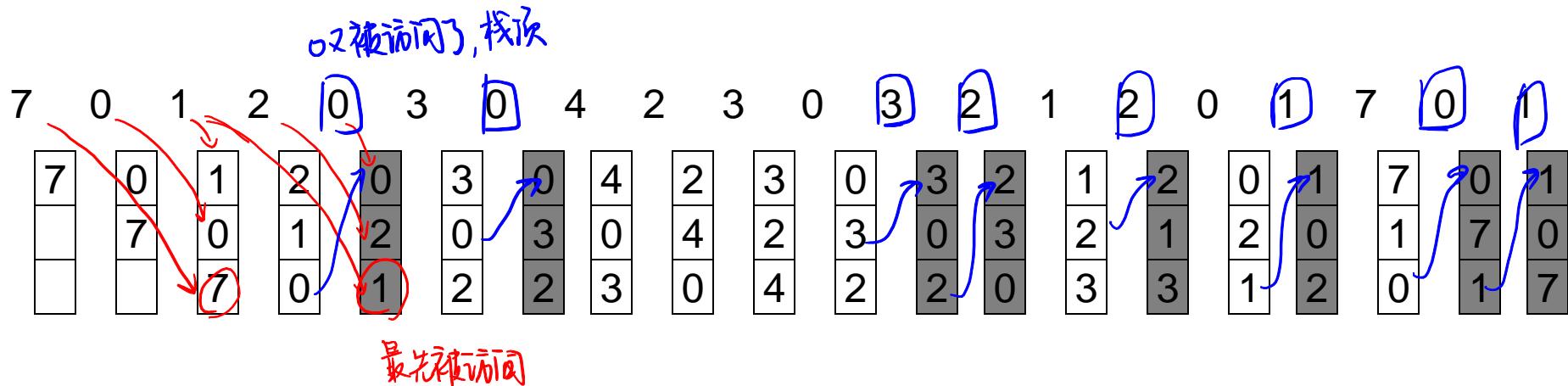
不管被访问时间

- 淘汰最先调入主存的页面，即淘汰在主存中驻留时间最长的页面。
- 实现方式
 - 采用循环数组实现
 - 实现复杂度低



LRU页面替换算法

- 淘汰的页面是最近一段时间内最久未被访问的那一页
- 基于程序局部性原理，即刚被使用过的页面可能还要立即被使用



表示未发生缺页中断和页面替换

LRU实现方法

1. 计数器实现

- 在页表项中增加一个 time-of-use 字段，并为 CPU 增加一个逻辑计数器。
- 每次内存访问，逻辑计数器加 1。
- 每次访问页面时，计数器寄存器的值拷贝到被访问页面的页表项中的 time-of-use 字段
- 需要替换时，替换 time-of-use 值 最小 的页面 越小，最近被访问。
- 每次页面替换需要查找页表，并且，每次内存访问都需要写内存一次（写页表的 time-of-use 字段）

2. 堆栈实现

双向链表

- 将页号放进堆栈
- 每次访问一个页面时，将其从栈中移到栈顶
- 最近最少使用的页面总是位于栈底
- 可以使用 双向链表 来实现栈的操作
- 无需查找页表，但每次页面更新需要 6 次指针改变操作。

copy

指向

近似LRU算法

reference bit.



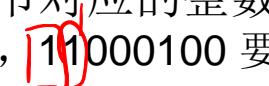
未被引用 \Leftarrow 淘汰

被引用

- 引用位法(最近未使用页面替换算法 NRU)

- 页表中的每一项都包含一个引用位, 起初设为0。当页面被访问时, 引用位被置为1。固定内存访问时间
- 间隔时间 t , 周期性地将所有页面的引用位清0。
- 页面置换时, 从引用位为0的页中挑选页面进行淘汰
- 选中要淘汰的页面后, 也将其他页的引用位清0。

- 额外引用位法

- 为每个页面维护额外的引用位状态码, 例如8位长的字节 
- 设置一个固定的定时器, 在定时器时间间隔内发生了页面的引用, 则将页面对应的引用位置1。
- 每次定时器中断后, 控制交给操作系统。
- 操作系统将该字节整体右移1位, 并将每个页面的引用位的值拷贝到8位字节的高位 $\gg 1$
- 将所有页面对应的引用位清0。
- 8位字节对应的整数值最小的页面可以被认为是最近最少使用的页面
 - 如,  要比  更近被使用

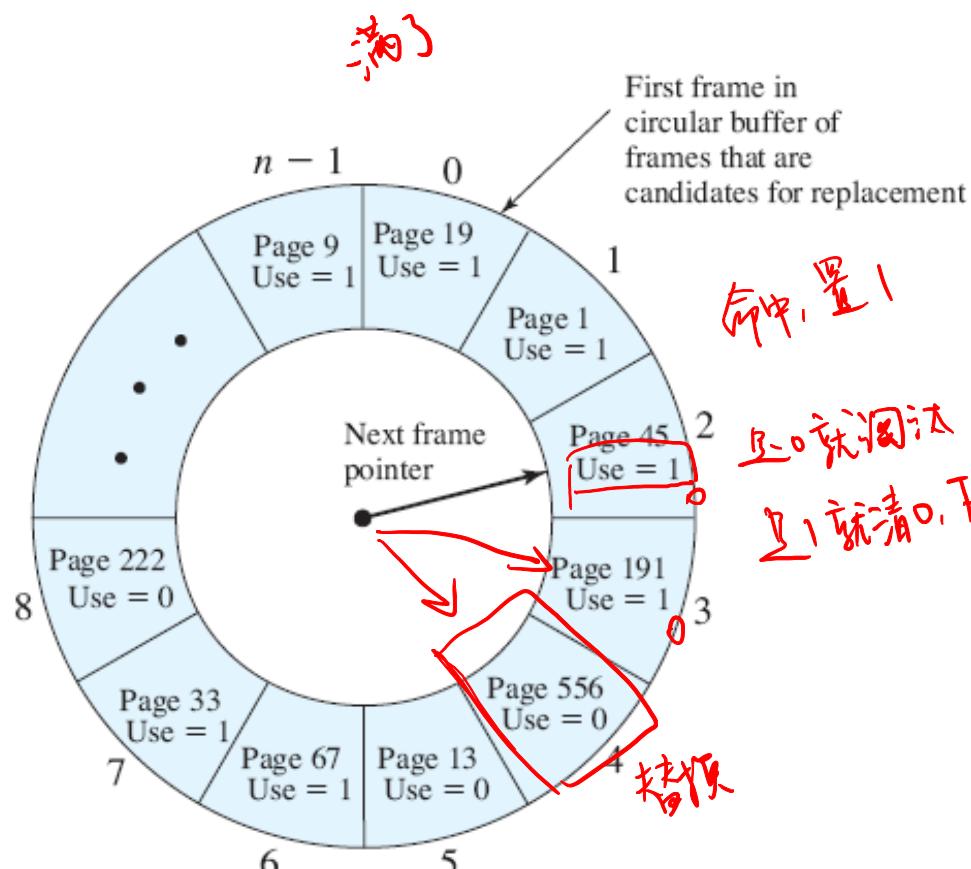
初期

越大 最近最少使用

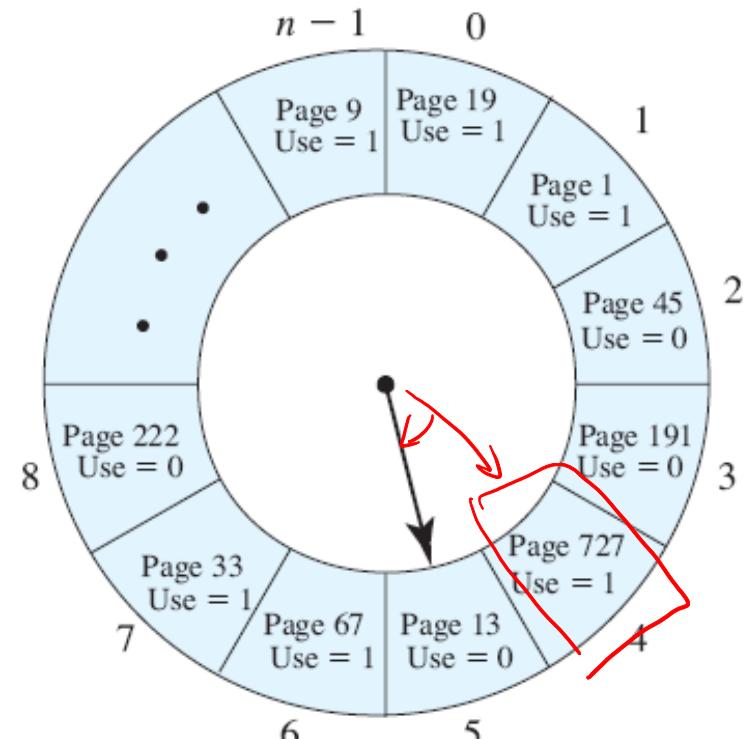
时钟页面替换算法

- 思路
 - FIFO易于实现，但不考虑程序的局部访问特性
 - LRU考虑程序的局部访问特性，但实现复杂度较高
 - 时钟页面替换算法是介于FIFO和LRU之间的一种方法
 - 利用主存中的引用位，替换时优先考虑引用位为0(即未被引用)的页面
- 实现方式
 - 进入主存的页面构造成循环队列
 - 指针给出了下一个可能被淘汰的候选对象
 - 主存中的任何一个页面被访问时，其引用位置1
 - 淘汰页面时，指针从当前位置指向的页面开始扫描循环队列，把遇到的引用位为1的页面的引用位清0，并继续向前扫描，如遇到引用位为0的页面，则淘汰该页面，指针指向下一个页面

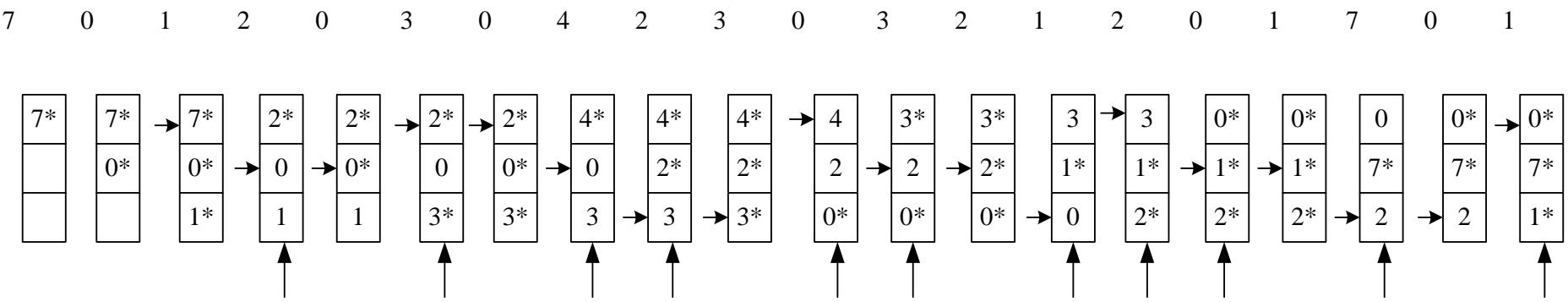
时钟页面替换算法的一个例子



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement



*表示页面的引用位为1

0

访问

修改

时钟页面替换算法的改进

被修改

未被修改 ✓

- 淘汰页面时，如果此页面已被修改过，必须将它先重新写回磁盘；但如果所淘汰的是未被修改过的页面，则无需将其写回磁盘；
- 淘汰修改过的页面比淘汰未被修改过的页面代价大；
- 可以将页表项的“引用位”和“修改位”结合起来使用，改进时钟页面替换算法

最近性

页面分为四种情况：

- ① 最近未被引用，未被修改($r=0, m=0$)
- ② 最近未被引用，被修改($r=0, m=1$)
- ③ 最近被引用，未被修改($r=1, m=0$)
- ④ 最近被引用，被修改($r=1, m=1$)

reference / modify :

局部性原理 .

最近性

时钟页面替换算法的改进

- 改进的替换算法
 - ① 从指针当前位置开始扫描循环队列，扫描过程中不改变“引用位”，把遇到的第一个 $r=0, m=0$ 的页面作为淘汰页 第一圈。
② 若步骤1失败，指针再次回到起始位置，查找 $r=0, m=1$ 的页面，把遇到的第一个这样的页面作为淘汰页，在扫描过程中把指针经过的页面的引用位 r 置0 第二圈。
③ 若步骤2失败，指针再次回到起始位置，由于此时所有页面的引用位 $r=0$ ，此时再转向步骤1或2，则一定能挑出一个可淘汰的页面 第三圈。
 - 也称为第三次机会时钟替换算法

替换算法性能比较

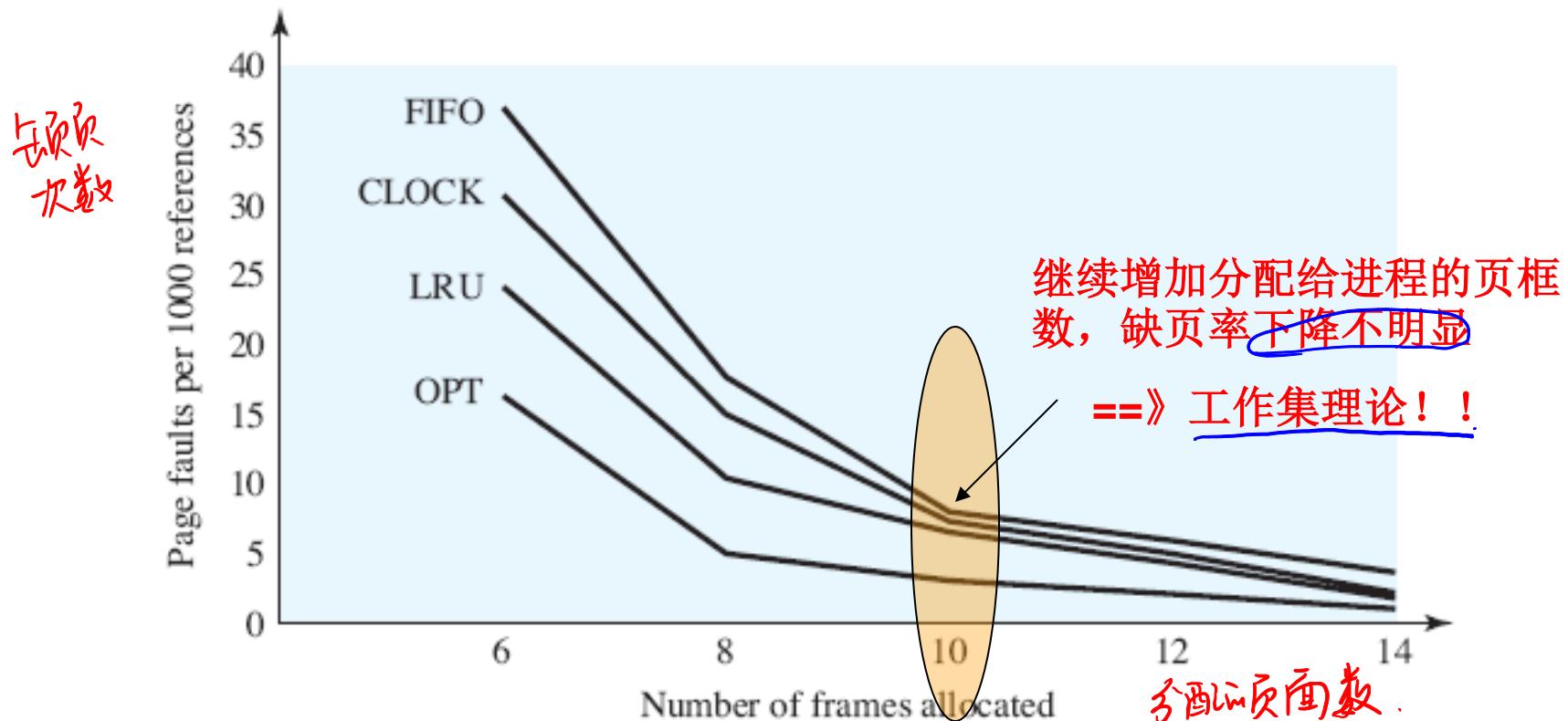


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

工作集理论

- 驻留集
 - 进程位于主存的页面集合
- 工作集理论及模型
 - 在某一段时间窗口内进程运行所需访问的页面集合
 - 时间窗口如何计量？以页面访问事件的发生来定义
 - 时间窗口多大？决定了工作集大小的上限
 - 监视每个进程的工作集，只有属于工作集的页面才能驻留在内存
 - 定期地从工作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；

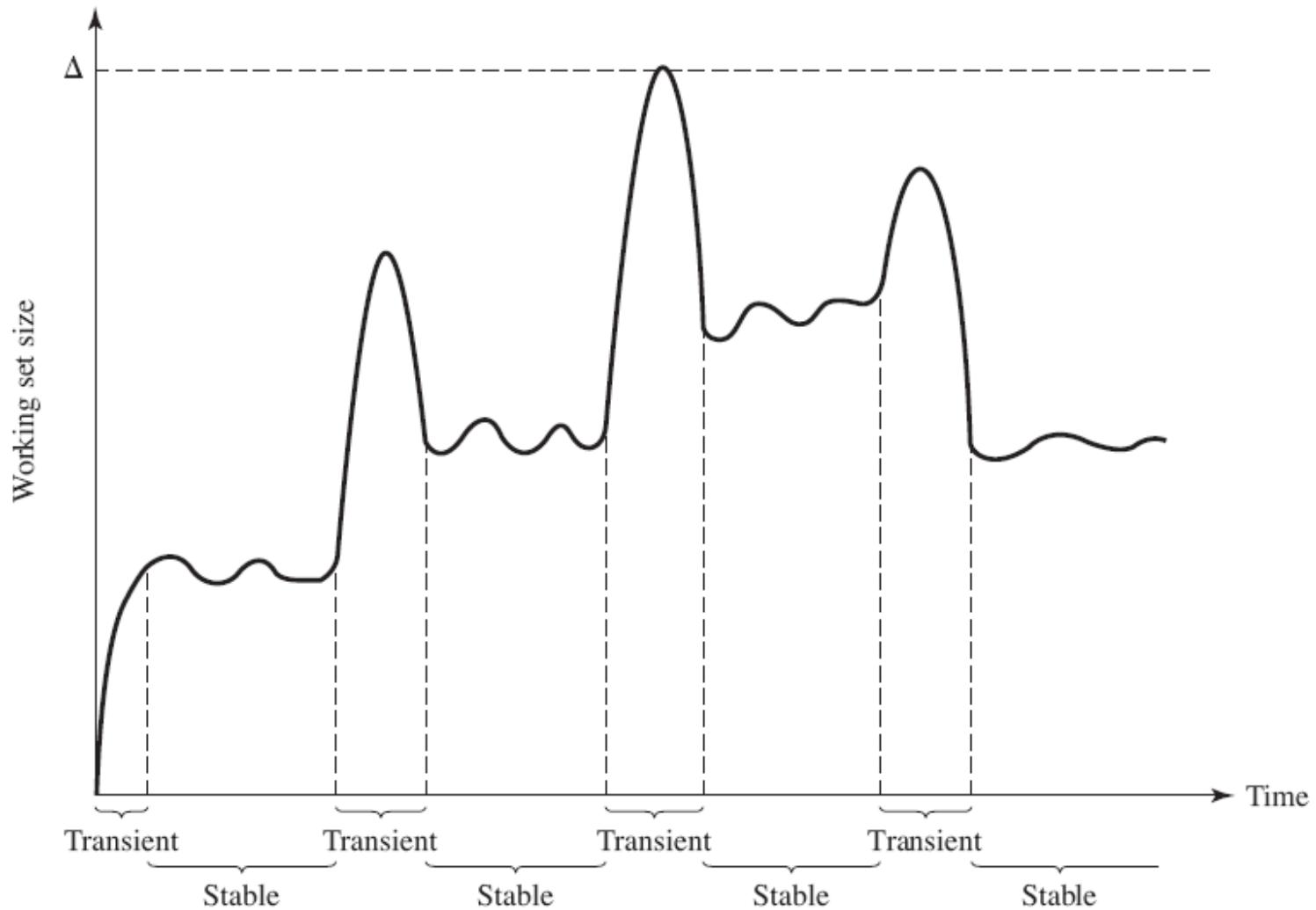
工作集理论

- 工作集是一个~~变化的集合~~
 - 是观察窗口大小的非减函数
 - 也是时间的函数
- 记~~W(t,Δ)~~表示时刻~~t-Δ~~到时刻t之间所访问的页面集合, ~~Δ~~为工作集窗口尺寸 (注意: 有些教材将 $W(t, \Delta)$ 定义为 $t-\Delta+1$ 到t之间所访问的页面集合)
 - Δ 太小, 将不包含整个locality; **局部性**
 - Δ 太大, 将包含多个locality;
 - 若 $\Delta = \infty$, 将包含整个程序.
- 设进程的~~页面总数为N~~, 则工作集大小满足:
$$1 \leq |W(t, \Delta)| \leq \min(\Delta+1, N)$$
$$|W(t, \Delta)| \subseteq |W(t, \Delta+1)|$$

工作集与窗口大小的关系

工作集	观察窗口 Δ			
	$[t-\Delta, t]$	1	2	$\Delta+1$ 重叠
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	-- 不变	--
17	24 17	23 24 17	18 23 24 17	24 15 18 23 17
18	17 18	24 17 18	-- 不变	18 17 23 24
24	18 24	--	17 18 24	--
18	--	18 24	--	17 18 24
17	18 17	24 18 17	--	--
17	17	18 17	--	--
15	17 15	17 15	15 17 18	15 17 18 24
24	15 24	17 15 24	15 17 24	--
17	24 17	--	--	15 17 24
24	--	24 17	--	--
18	24 18	24 17 18	17 18 24	15 17 18 24

工作集与时间的关系



局部最佳页面替换算法

- 与全局最佳替换算法类似，需要预知程序的页面访问序列
- 如果页面在 $(t, t+\Delta)$ 窗口内将被访问，则页面留在驻留集， t 为当前时刻， $(t, t+\Delta)$ 为滑动窗口
窗口页框数固定

某

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	(P3)	(P3)	P4	(P2)	(P3)	(P5)	(P3)	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	✓	-
P2	-	-	-	-	✓	访问	-	-	-	-	-
P3	-	✓	访问	✓	✓	访问	✓	✓	-	-	-
P4	✓	访问	✓	✓	-	访问	-	-	-	-	✓
P5	-	-	-	-	-	-	✓	✓	✓	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

窗口内

被访问的页
齐祖 \Rightarrow 高生

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	√	-
P2	-	-	-	-	√	-	-	-	-	-	-
P3	-	√	√	√	√	√	√	√	-	-	-
P4	√	√	√	√	-	-	-	-	-	-	√
P5	-	-	-	-	-	-	√	√	√	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	√	-
P2	-	-	-	-	√	-	-	-	-	-	-
P3	-	√	√	√	√	√	√	√	-	-	-
P4	√	√	√	√	-	-	-	-	-	-	√
P5	-	-	-	-	-	-	√	√	√	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	√	-
P2	-	-	-	-	√	-	-	-	-	-	-
P3	-	√	√	√	√	√	√	√	-	-	-
P4	√	√	√	√	-	-	-	-	-	-	√
P5	-	-	-	-	-	-	√	√	√	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	√	-
P2	-	-	-	-	√	-	-	-	-	-	-
P3	-	√	√	√	√	√	√	√	-	-	-
P4	√	√	√	√	-	-	-	-	-	-	√
P5	-	-	-	-	-	-	√	√	√	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	√	-
P2	-	-	-	-	√	-	-	-	-	-	-
P3	-	√	√	√	√	√	√	√	-	-	-
P4	√	√	√	√	-	-	-	-	-	-	√
P5	-	-	-	-	-	-	√	√	√	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	√	-
P2	-	-	-	-	√	-	-	-	-	-	-
P3	-	√	√	√	√	√	√	√	-	-	-
P4	√	√	√	√	-	-	-	-	-	-	√
P5	-	-	-	-	-	-	√	√	√	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	√	-
P2	-	-	-	-	√	-	-	-	-	-	-
P3	-	√	√	√	√	√	√	√	-	-	-
P4	√	√	√	√	-	-	-	-	-	-	√
P5	-	-	-	-	-	-	√	√	√	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

局部最佳页面替换算法示例

滑动窗口 $\Delta=3$

工作集替换算法

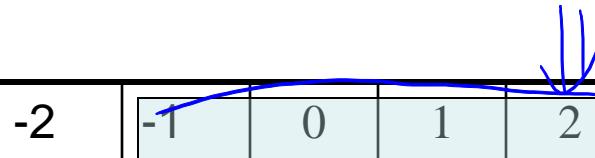
- 最佳页面替换算法的问题：
 - 需要预知未来的页面访问序列，不现实！
- 解决办法：
 - 基于历史信息预测未来
 - 进程不久的将来所访问的页面集合可通过考查其最近时间内的主存需求做出估计
- 在 t 时刻，如果页面在 $(t-\Delta, t)$ 窗口内被访问过，则该页面留在工作集，驻留内存

历史

↓

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$



时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$

The diagram shows a table illustrating the working set replacement algorithm. The columns represent time steps from -2 to 10. The rows represent pages P1 through P5, and external events In_t and Out_t. The table entries indicate whether a page is present (✓) or absent (--) in a working set at a given time. A blue circle highlights the sequence of pages P1, P3, P3, and P4 from time t=0 to t=3. A blue bracket groups P3 and P4. A blue arrow points down to the row for page P3 at t=3. A blue brace groups the four rows for pages P1, P3, P4, and P5 at t=3. A blue curly brace groups the four rows for pages P1, P3, P4, and P5 at t=4.

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			✓	✓	✓	✓ l	--	--	--	--	--	✓	✓
P2			--	--	--	--	✓	✓	✓	✓	--	--	--
P3			--	✓	✓	✓ 3	✓	✓	✓	✓	✓	✓	✓
P4			✓	✓	✓	✓ 4	✓	✓	✓	--	--	--	✓
P5			✓	✓	--	--	--	--	✓	✓	✓	✓	✓
In _t			P3				P2		P5			P1	P4
Out _t				P5		P1			P4	P2			

工作集替换算法示例， $\Delta=3$

↓ t=4

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3		P2		P5			P1		P4
Out _t					P5	P1			P4	P2			

工作集替换算法示例， $\Delta=3$

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问 页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
访问页面	P5	P4	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1			√	√	√	√	--	--	--	--	--	√	√
P2			--	--	--	--	√	√	√	√	--	--	--
P3			--	√	√	√	√	√	√	√	√	√	√
P4			√	√	√	√	√	√	√	--	--	--	√
P5			√	√	--	--	--	--	√	√	√	√	√
In _t				P3			P2		P5			P1	P4
Out _t					P5		P1			P4	P2		

工作集替换算法示例， $\Delta=3$

工作集替换算法

- 纯工作集替换算法的挑战：
 - 需要为每个进程监督页面的变化，开销较大
 - 估算合适的 Δ 值也是个难题
- 改进方案：
 1. 模拟工作集替换算法
 2. 缺页频率替换算法

模拟工作集替换算法

- 间隔定时器+引用位+年龄寄存器
- 每隔时间 t , 系统扫描主存中的所有页面, 先将年龄寄存器右移一位, 然后将引用位的值加到年龄寄存器的最左边.
- 若年龄寄存器的值为0, 则此页面移出工作集
- 例, 年龄寄存器4位, 时间间隔 t 为1000个指令周期, 页面在4000条指令内都未再被引用, 则:

$$\begin{aligned} R(0) &= 1000, R(1000) = 0100, R(2000) = 0010, \\ R(3000) &= 0001, R(4000) = 0000 \end{aligned}$$

copy

4bit

4个周期内都未被访问

缺页频率替换算法

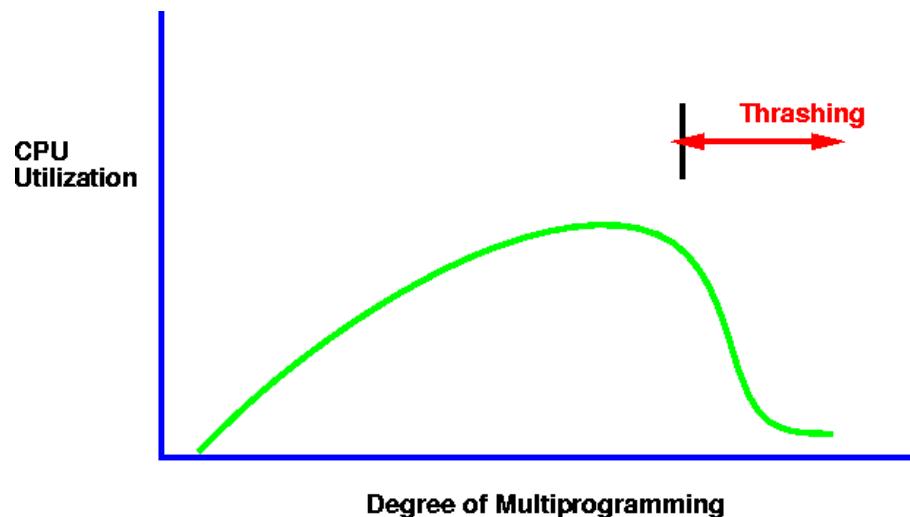
- 工作集替换算法中，**每次引用页面都可能引起工作集的变化**
- 调入新页面
- 老页面不再位于工作集内
 - 缺页频率替换算法
 - 依据缺页率动态调整工作集大小
 - **仅在缺页中断发生时才调整工作集大小** 命中时不调整
 - 定义一个临界时间间隔值 Γ
 - 每次缺页时，若 **本次缺页与前次缺页之间的时间间隔** 小于 Γ ，则为该进程增加一个驻留内存的新页框；
缺页频率 \leftarrow **frame + 1**
 - 否则，将在这个时间**间隔**内未被引用的所有页面移出内存
- 上次缺页 这一次缺页之间

抖动(trashing)

- 若进程分配的页面过少, 则缺页率将非常高, 将导致:
 - CPU利用率低
 - OS认为应增加程序的道数
 - 另一个进程加入系统
 - 系统吞吐量骤降...
 - 抖动
 - 进程忙于在内存和外存之间对换页面, 花在页面对换的时间远大于实际执行的时间

抖动

- $WSS_j(t)$ 表示进程 P_j 在 t 时刻的工作集尺寸
- $D = \sum_j WSS_j(t) \equiv$ 需求的总页框数
 - 若 $D > m$ (总的可用页框数) \Rightarrow 抖动
- 抖动将急剧恶化CPU的利用率 ↓
进程太多，页框数太少。
- 策略：
 - 如果 $D > m$, 则 挂起一个进程
 - 选择哪个进程挂起?



请求分页管理中的几个问题

- 页面大小
 - 控制页表占用的额外内存角度, 大页面好 \Rightarrow 负载少
 - 降低内部碎片大小, 小页面好
 - 从提高I/O效率, 大页面好
- 典型的页面大小
 - 512B~8K不等

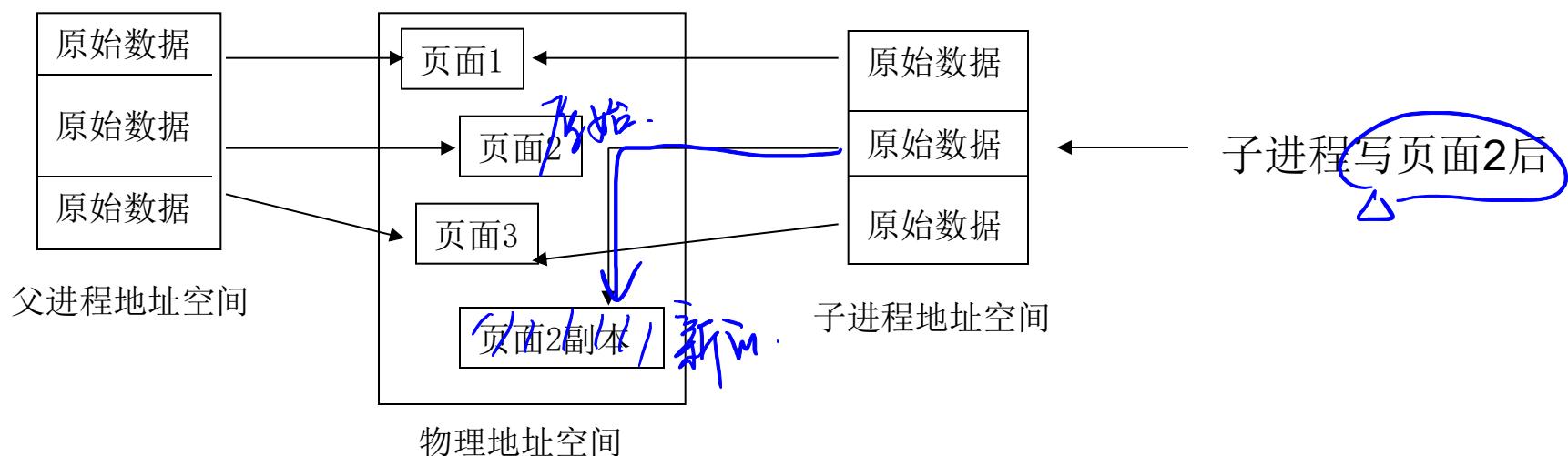
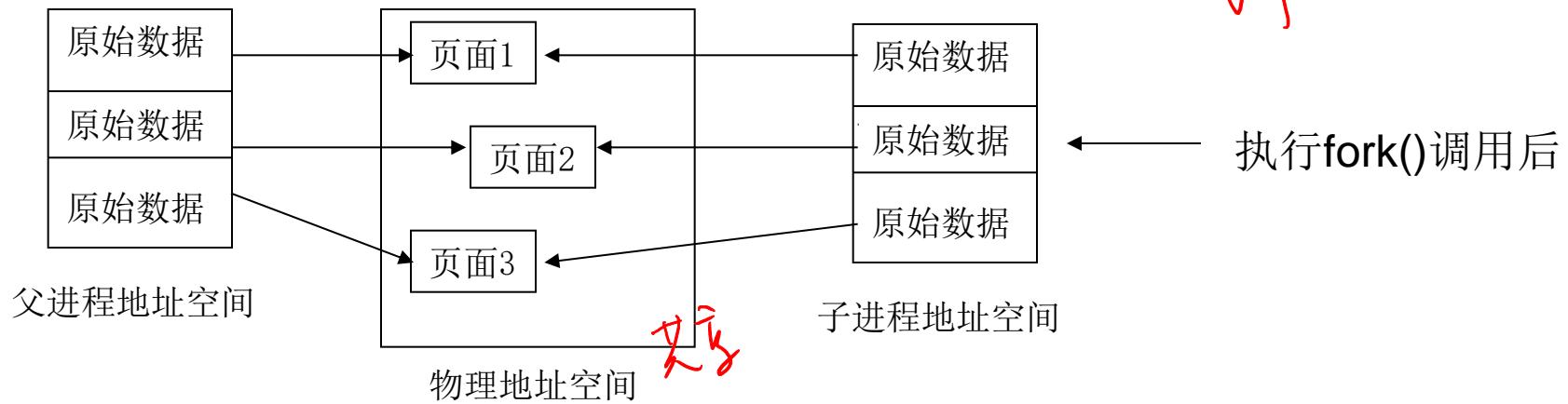
请求分页管理中的几个问题

- 页面交换区
 - 系统初始化时，保留一定的磁盘空间作为页面交换区，不能被文件系统使用，与主存一起构成虚拟存储空间
 - 进程创建时，预留出与进程大小一样的交换空间，进程撤销时，释放占用的交换空间
 - 外页表记录进程页面号和磁盘块号的对应

请求分页管理中的几个问题

- 写时复制
 - 写时复制(copy-on-write)是存储管理节省物理主存(页框)的一种页面级优化技术，已被UNIX和Windows等采用，能减少主存页面内容的复制操作，减少相同内容页面在主存的副本数目。
 - 进程创建时并不复制父进程的完整空间，而仅复制父进程的页表，使父子进程共享物理空间，并将共享空间的访问权限设为“只读” 不修改之后会被父亲看到。
 - 当其中某个进程要修改页面内容执行写操作时，会产生“写时复制”中断
 - 操作系统处理“写时复制”中断，为此进程创建一个新页，设置其为可读可写，并将其映射到进程的地址空间，此进程就可以修改复制的页
 - 采用写时复制使得地址空间复制的操作可以推迟到修改时 临时内存复制而工作

vfork() 共享 + data



写时复制示意图

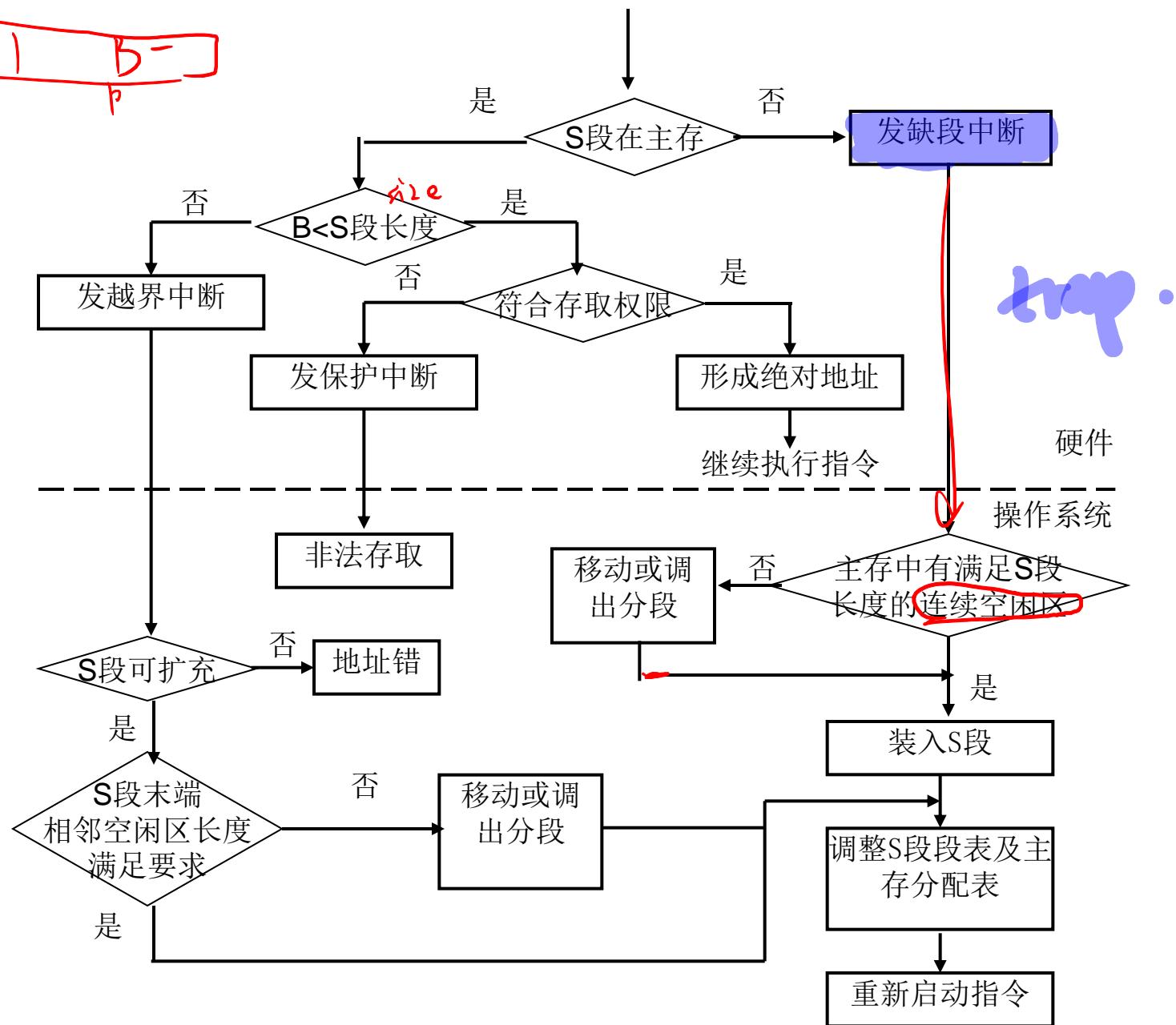
大纲

- 虚拟内存的概念
- 请求分页虚拟存储管理
- 请求分段虚拟存储管理
- 请求段页式虚拟存储管理

请求分段虚拟存储管理

- 作业的所有分段的副本都存放在辅存中
- 当作业调度运行时，首先把当前需要的段装入内存，在执行过程中访问到不在内存的段时再将其动态装入
- 当所需的段不在主存中时，触发**缺段中断**
- 段表中需要增设供管理使用的若干表项
 - 特征位
 - 存取权限
 - 扩充位
 - 标志位
 - . . .

S | B-
P



大纲

- 虚拟内存的概念
- 请求分页虚拟存储管理
- 请求分段虚拟存储管理
- 请求段页式虚拟存储管理

请求段页式存储管理

- 虚地址以程序的逻辑结构划分成段
- 实地址划分成固定长度的页框
- 将每一段的线性地址空间划分成页框大小一样的页面
- 逻辑地址分为三部分： 段号、段内页号、
页内偏移
- 地址转换过程与段页式存储管理类似，加入了缺段中断和缺页中断处理

本章小结

- 虚拟存储器使得无需将进程全部装入内存就可以执行程序，其余部分存放在磁盘，在执行过程中按需装入
- 程序执行的局部性原理保证虚拟存储器的效率
- 请求分页/段式存储管理通过缺页/段中断在指令的执行过程中载入所需的页面/段
- 页面分配策略包括固定分配和可变分配，前者在进程的生命周期中保持页框数不变，后者进程的页框数可变
- 全局替换指被替换的页面可以是任何进程的页面，局部替换指被替换的页面只能是属于本进程的页面；
- 页面替换算法包括OPT, FIFO, LRU, CLOCK等
- 工作集是指在某个时间间隔内访问的页面集，随着时间的变化而变化
- 为保证程序正常运行，需要将程序的工作集调入内存，当内存不能容纳所有进程的工作集时，会发生抖动现象。