操作系统A

Principles of Operating System

北京大学计算机科学技术系 陈向群

Department of computer science and Technology, Peking University

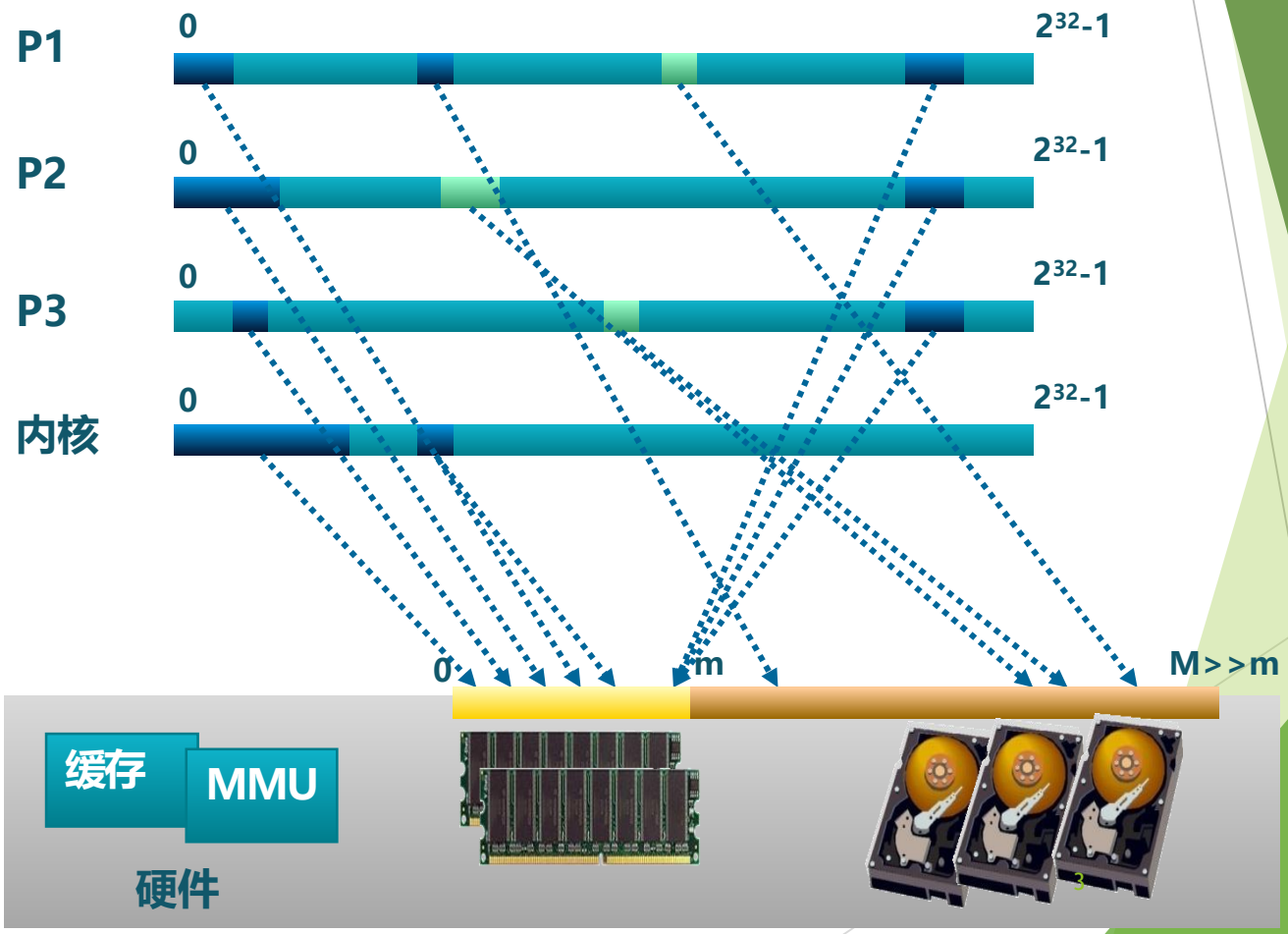
2020 Autumn

存储管理——大纲

- 重要概念
  - 存储体系、存储保护、地址重定位
- 物理内存管理
  - 数据结构（位示图、空闲区表、空闲区链表）
  - 分配算法（首先适配、最佳适配、最差适配 ……）
- 各种存储管理方案
  - 单一连续区、固定分区、可变分区、页式、段式、段页式
- 虚拟存储管理
  - 硬件、页表、页错误处理
  - 软件策略：读取策略、放置策略、置换策略、驻留集策略、清除策略、装载控制策略

操作系统的存储抽象

- 操作系统对存储的抽象：地址空间

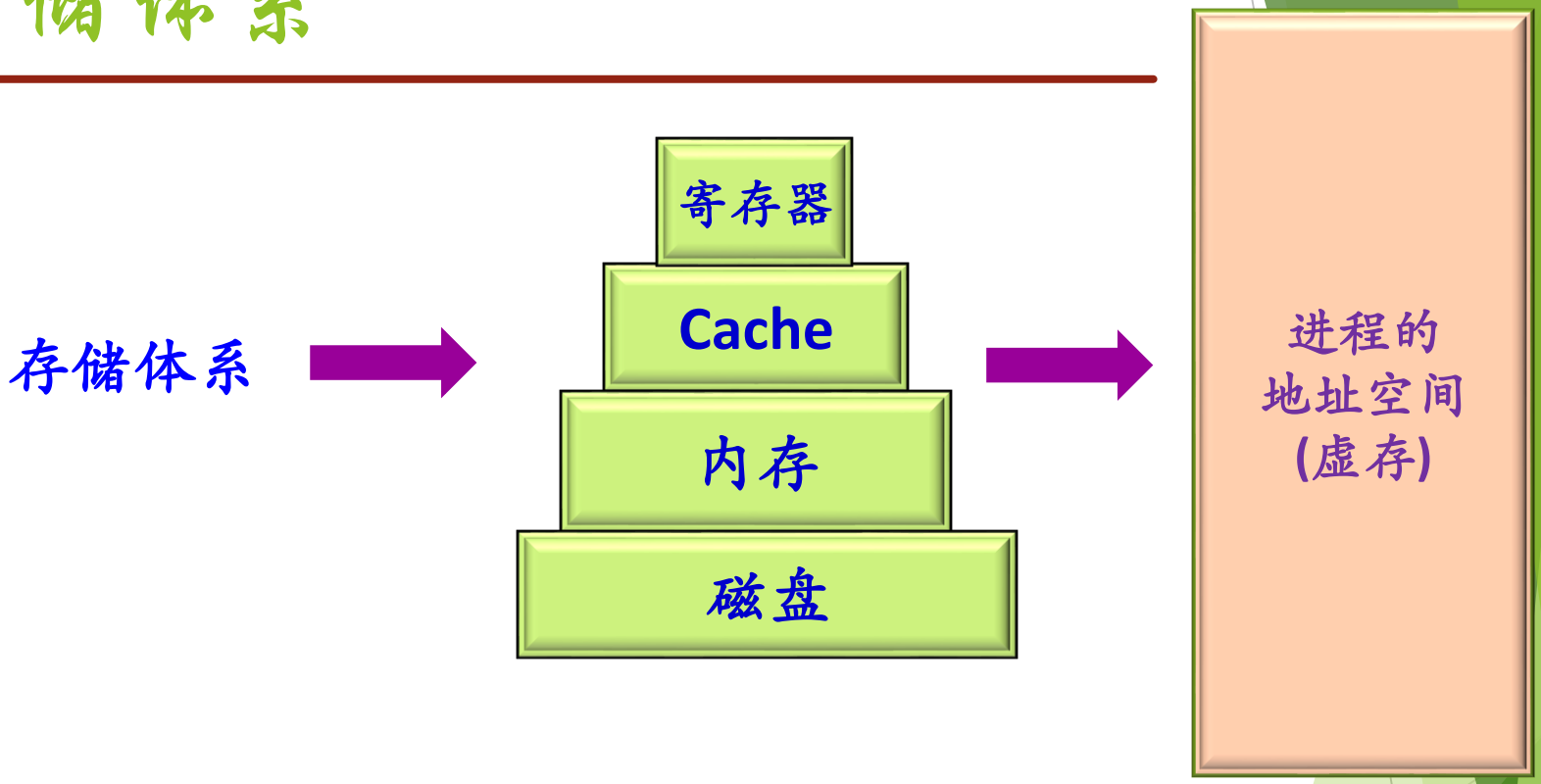


Acknowledgement: 清华大学 向勇、陈渝

基本思想、设计与实现问题、……

虚拟页式存储管理

存储体系



- 由操作系统协调各存储器的使用
- 目的：

“内存”速度尽量快，与CPU取指速度相匹配，
“内存”容量尽量大，能装下当前运行的程序与数据

相关术语辨识

- 虚拟内存
- 虚拟地址
- 虚拟地址空间
- 虚拟存储技术

*把物理内存与磁盘结合起来使用，得到一个容量很大的“内存”，即虚存
*程序引用内存所使用的地址与内存物理地址是不同的，可被自动转换成物理地址
*虚存大小受计算机系统寻址机制和可用磁盘容量的限制

虚拟内存中某一位置的地址，该位置可以被访问，仿佛它是内存的一部分

分配给进程的虚拟内存

当进程运行时，先将其一部分装入内存，另一部分暂时保存在磁盘；当要执行的指令或访问的数据不在内存时，由操作系统自动完成将它们从磁盘调入内存的工作

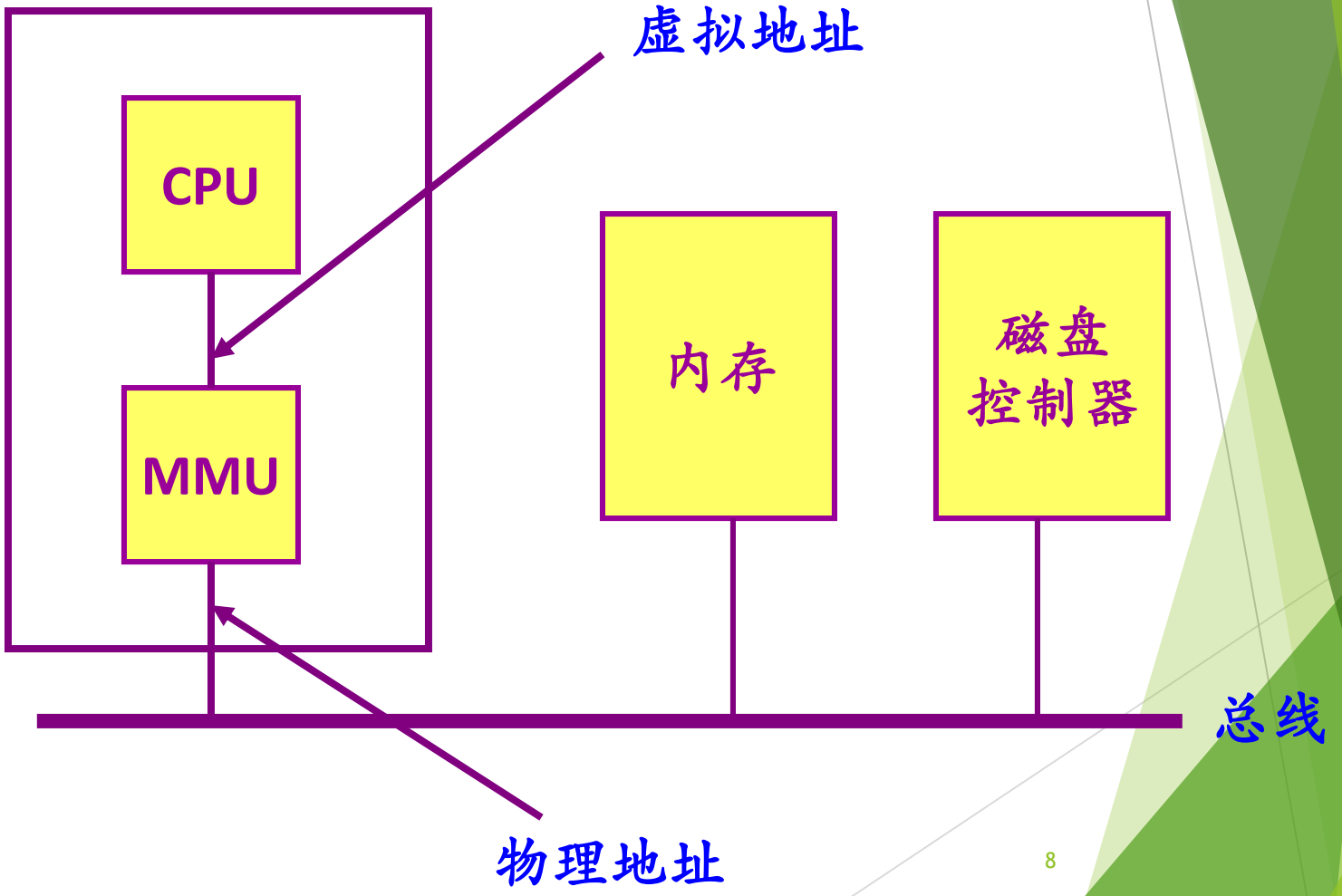
虚拟页式存储管理

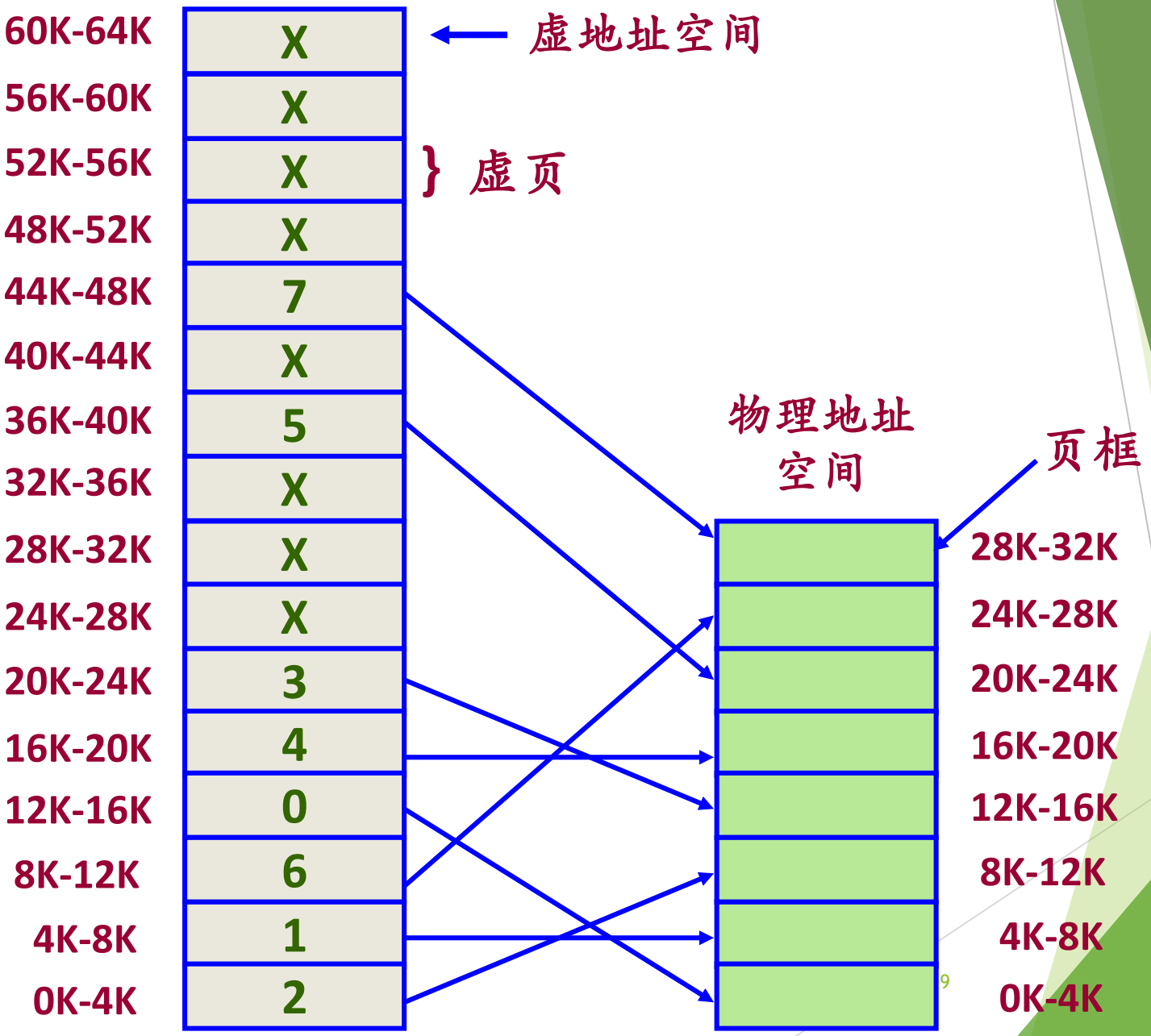
Page Fault

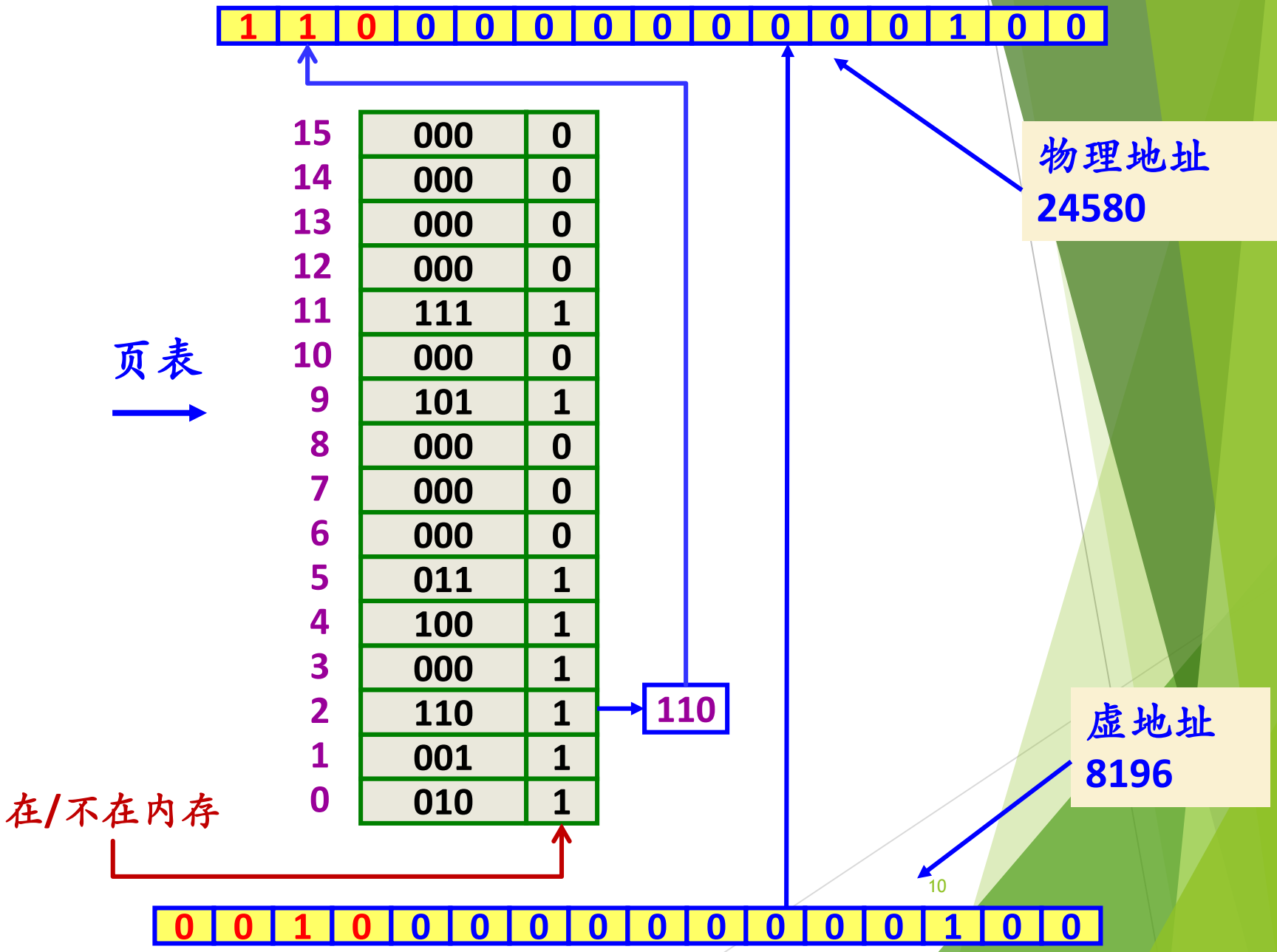
- 基本思想
  - 装载程序时，不是装入全部页面，而是装入几个甚至零个页面
  - 如果进程执行时需要的页面不在内存，则动态装入所需页面
  - 需要时，将内存中暂时不用的一些页面交换到磁盘，以便获得更多的内存空间
- 通常有两种方式
  - 请求调页（demand paging）
  - 预先调页（prepaging）

操作系统中的资源转换技术
——以CPU时间和磁盘空间换取物理内存空间

MMU:内存管理单元




60K-64K
56K-60K
52K-56K
48K-52K
44K-48K
40K-44K
36K-40K
32K-36K
28K-32K
24K-28K
20K-24K
16K-20K
12K-16K
8K-12K
4K-8K
0K-4K
X
X
X
X
7
X
5
X
X
X
3
4
0
6
1
2
虚地址空间
虚页
物理地址
空间
页框
28K-32K
24K-28K
20K-24K
16K-20K
12K-16K
8K-12K
4K-8K
0K-4K

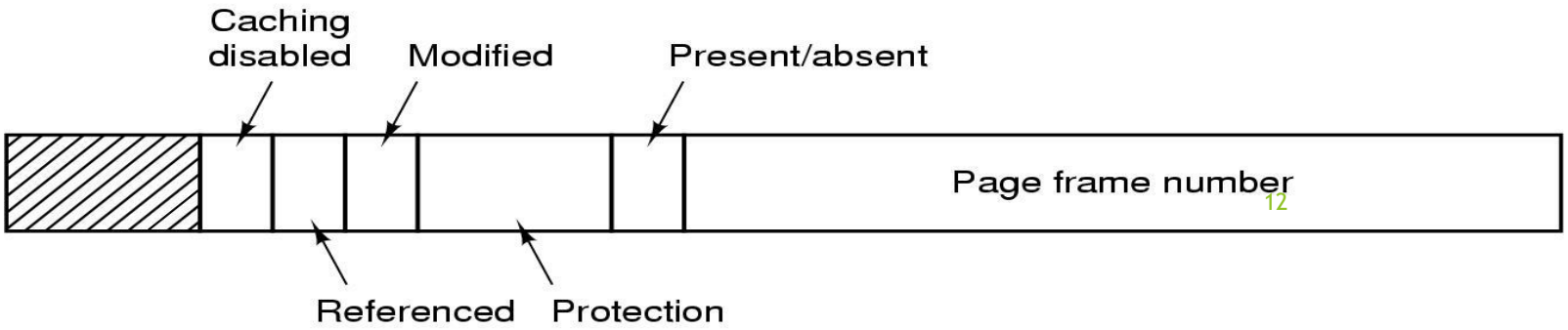

1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
物理地址
24580
页表
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
110
在/不在内存
虚地址
8196
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0

设计与实现时要解决的问题

- 页表表项的设计
- 如何处理页表巨大的问题?
- 地址重定位与快表(TLB)
- 一种最常见的Page Fault → 缺页中断
- 驻留集管理
- 置换策略
- 清除策略
- 加载控制

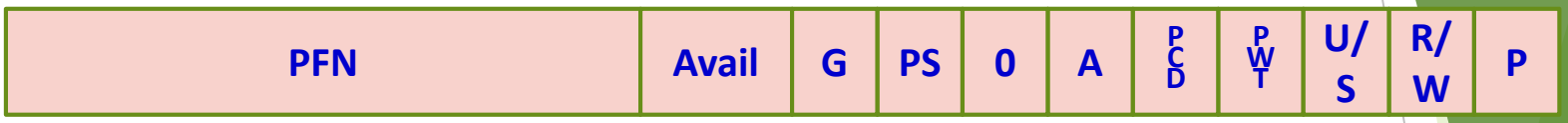
1. 页表表项设计

- 页框号、有效位、访问位、修改位、保护位等
  - ✓ 页框号（内存块号、物理页面号、页帧号）
  - ✓ 有效位（驻留位、中断位）：表示该页是在内存还是在磁盘（Valid、Present）
  - ✓ 访问位：引用位（Referenced、Accessed）
  - ✓ 修改位：查看此页是否在内存中被修改过（Dirty、Modified）
  - ✓ 保护位：读/写/执行（Protection）

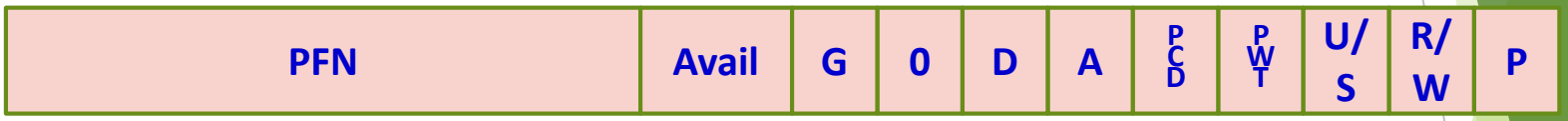


i386页目录项和页表项

页目录项 PDE (Page Directory Entry)

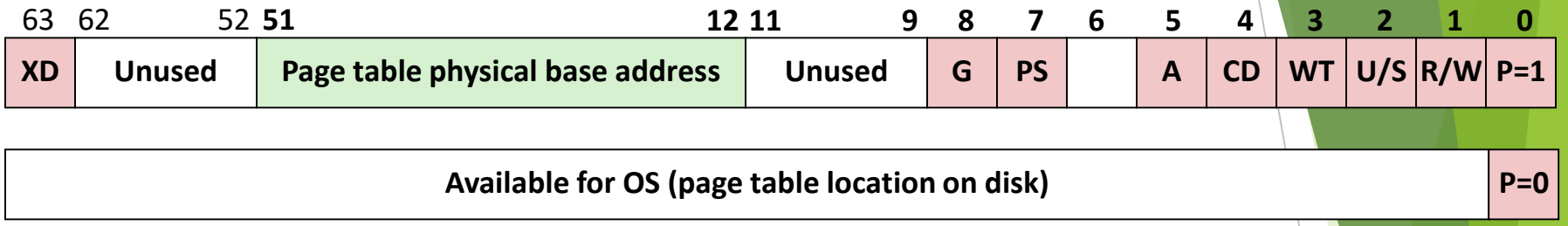


页表项 PTE (Page Table Entry)



PFN(Page Frame Number)：页框号
P(Present)：有效位
A(Accessed)：访问位
D(Dirty)：修改位
R/W(Read/Write)：只读/可读写
U/S(User/Supervisor)：用户/内核
PWT(Page Write Through)：缓存写策略
PCD(Page Cache Disable)：禁止缓存
PS(Page Size)：大页4M

Core i7 Level 1-3 Page Table Entries



Each entry references a 4K child page table

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

2. 多级页表

- 计算一下

  32位虚拟地址空间的页表规模?

  页面大小为 4K；页表项大小为 4 字节；

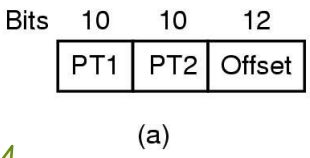
  用户地址空间为 2G = 2^{31}

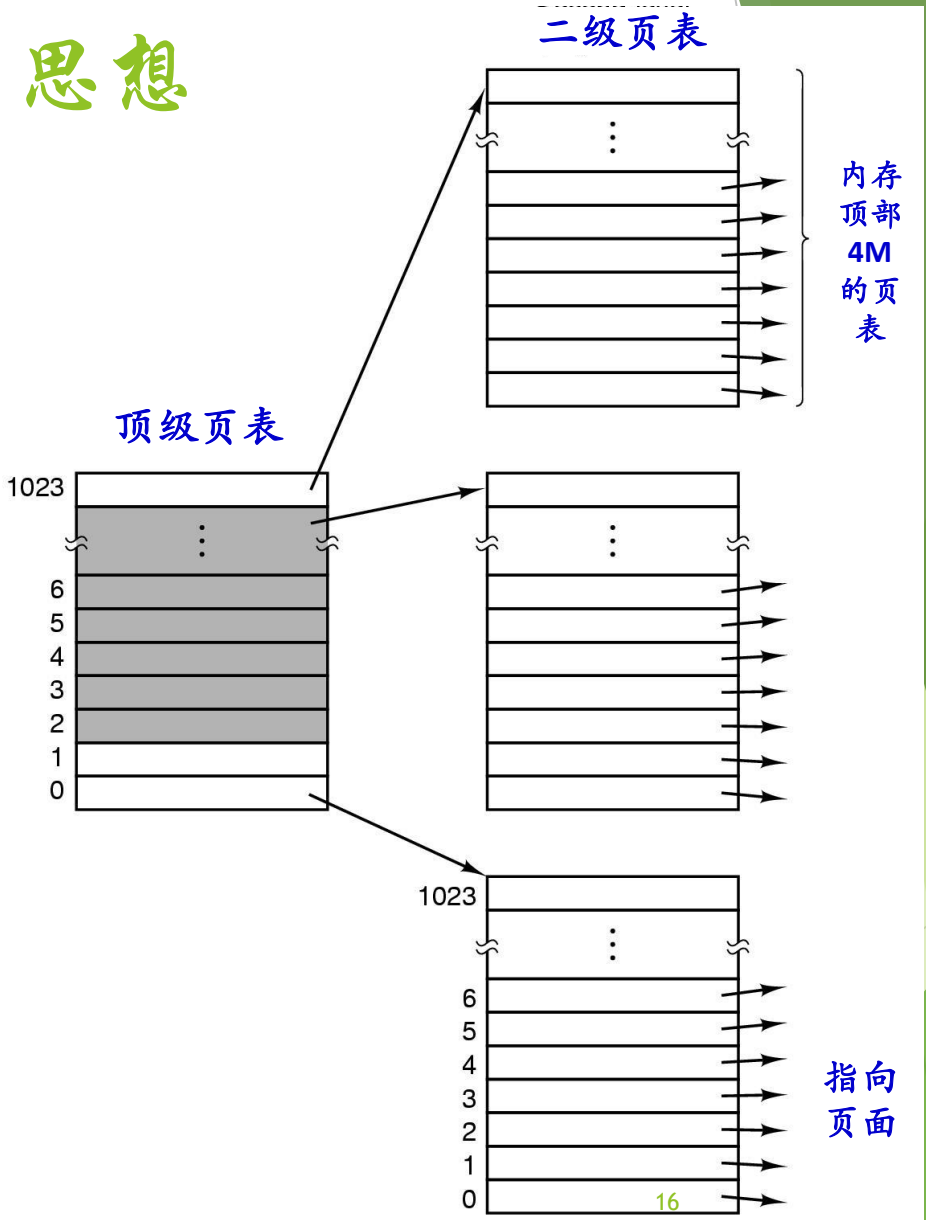
  则：一个进程有 ？ 页；其页表需要占 ？ 页

- 一个进程的页表的各页在内存中若不连续存放，则需要引入地址索引 → **页目录**

- 注意：页表本身也放在虚存中（进程运行时，部分页表映射到内存）

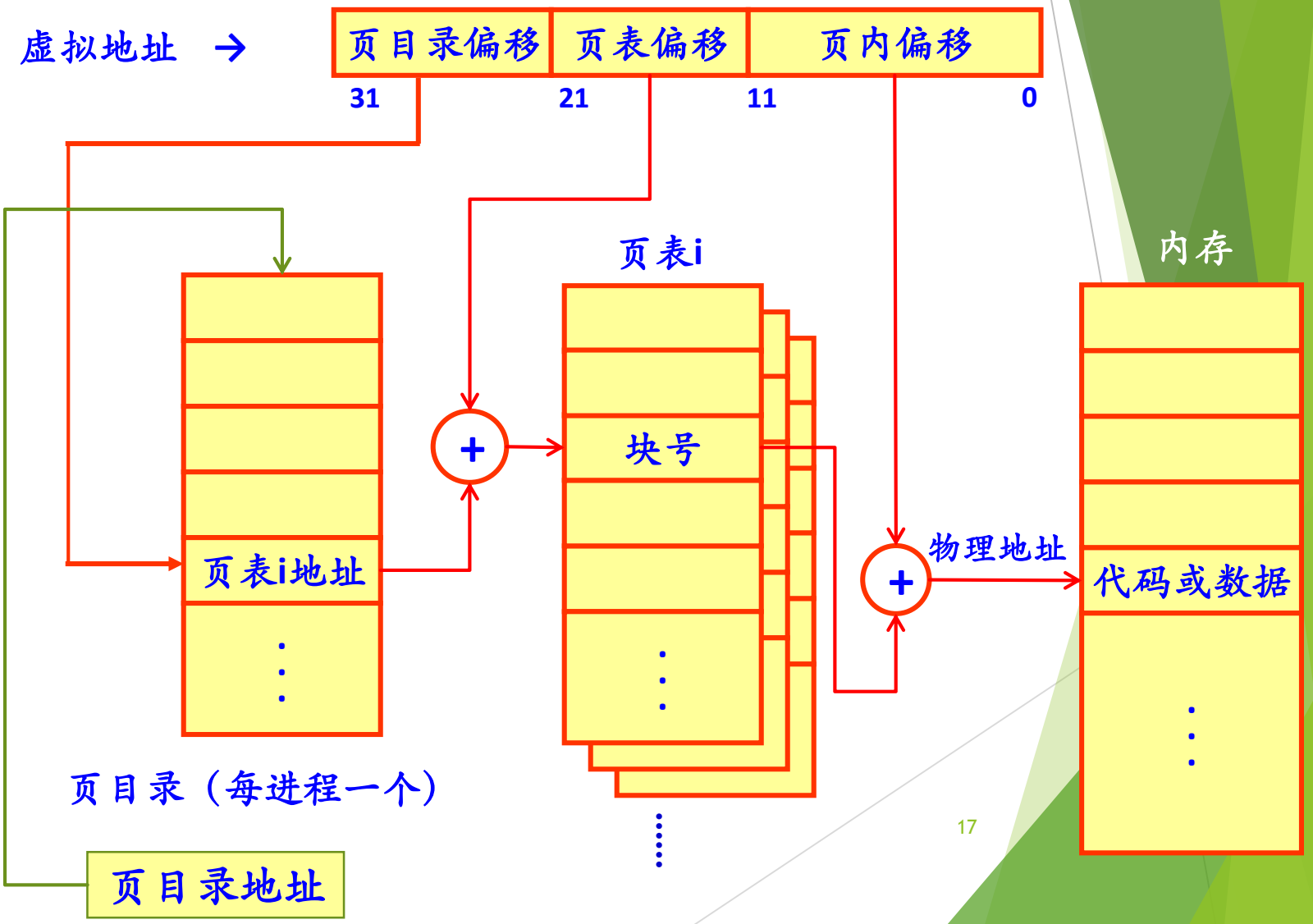
两级页表设计思想





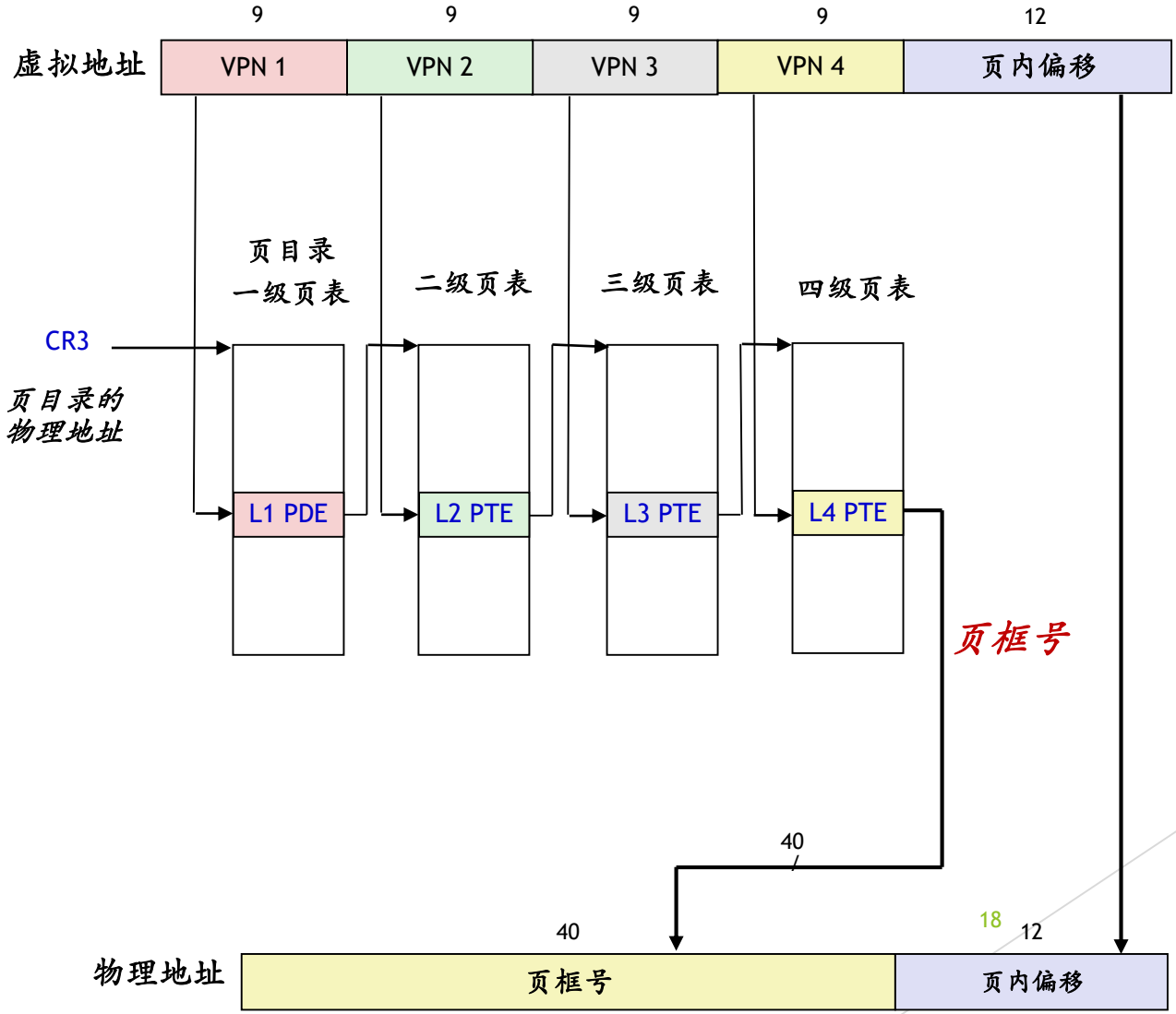
相当于数组转变为树形结构

二级页表结构及地址映射

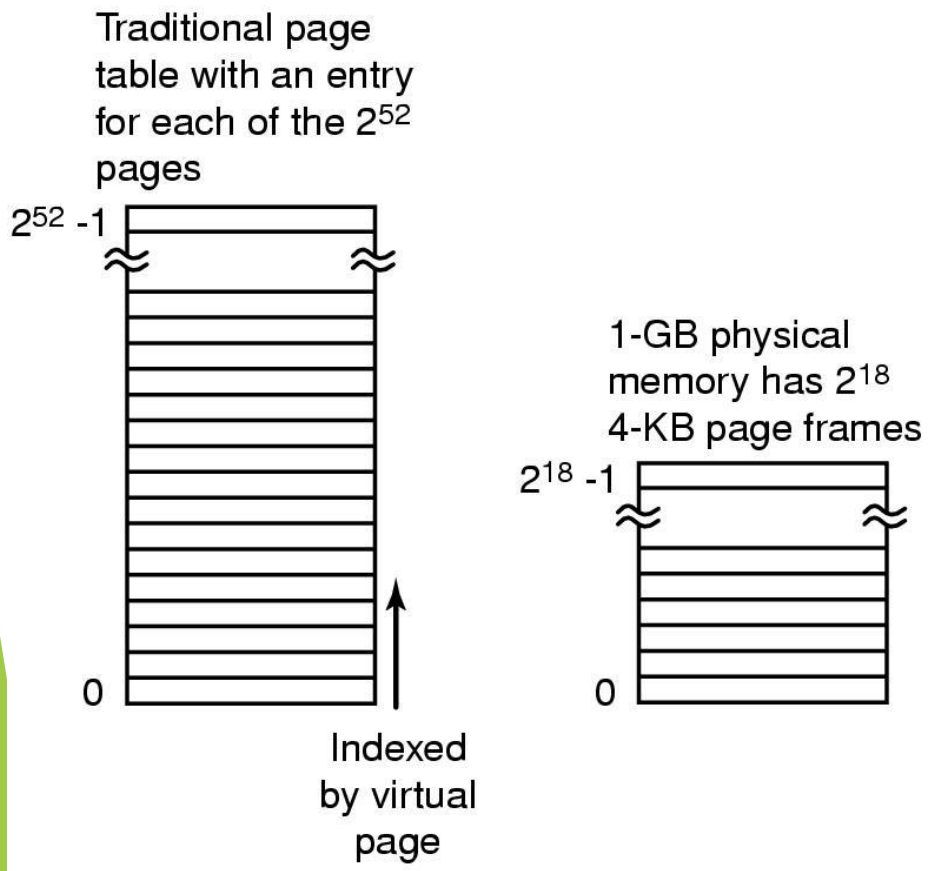


Core i7 页表结构

虚拟地址空间2^{48}



3.引入反转(反置、反向)页表



- 地址转换

-- 从虚拟地址空间出发：虚拟地址 → 查页表 → 得到页框号 → 形成物理地址

-- 每个进程一张页表

- 反转页表思路

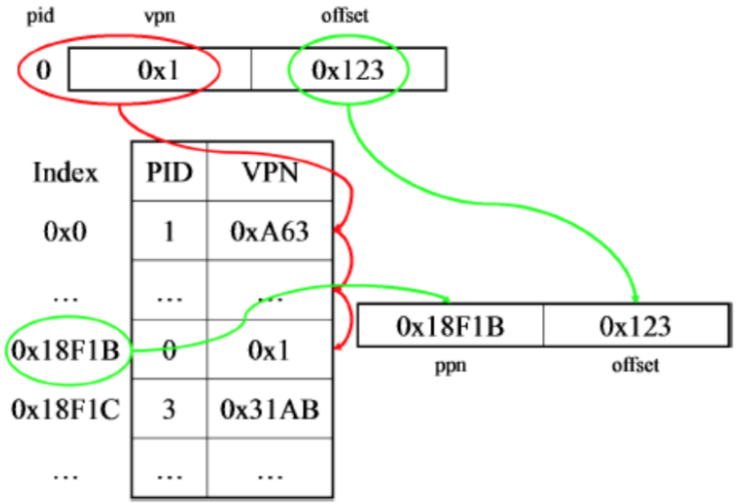
-- 从物理地址空间出发，系统建立一张页表

-- 页表项记录进程i的某虚拟地址(虚页号)与页框号的映射关系

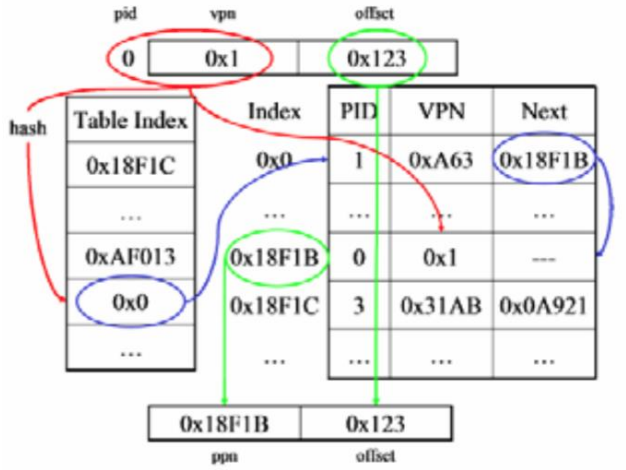
反转(反置、反向)页表示例

- 参考virginia的cs333

Accessing Inverted Page Table



Accessing Hashed Inverted Page Table



图片来源http://www.cs.virginia.edu/~cs333/notes/virtual_memory4.pdf

小结

思考题：调研某一个体系结构的反转页表机制

- PowerPC、UltraSPARC和IA-64 等体系结构采用

做法：

- 将虚拟地址的页号部分与进程pid映射到一个散列值
- 散列值指向一个反转页表
- 需要拉链解决冲突问题

- 反转页表大小与实际内存成固定比例，与进程个数无关

4. 地址转换

- 地址转换过程(硬件机制)

if (虚拟页面不在内存、页面非法、或者被保护) {

硬件产生异常，陷入操作系统，执行页面错误服务程序（Page Fault）

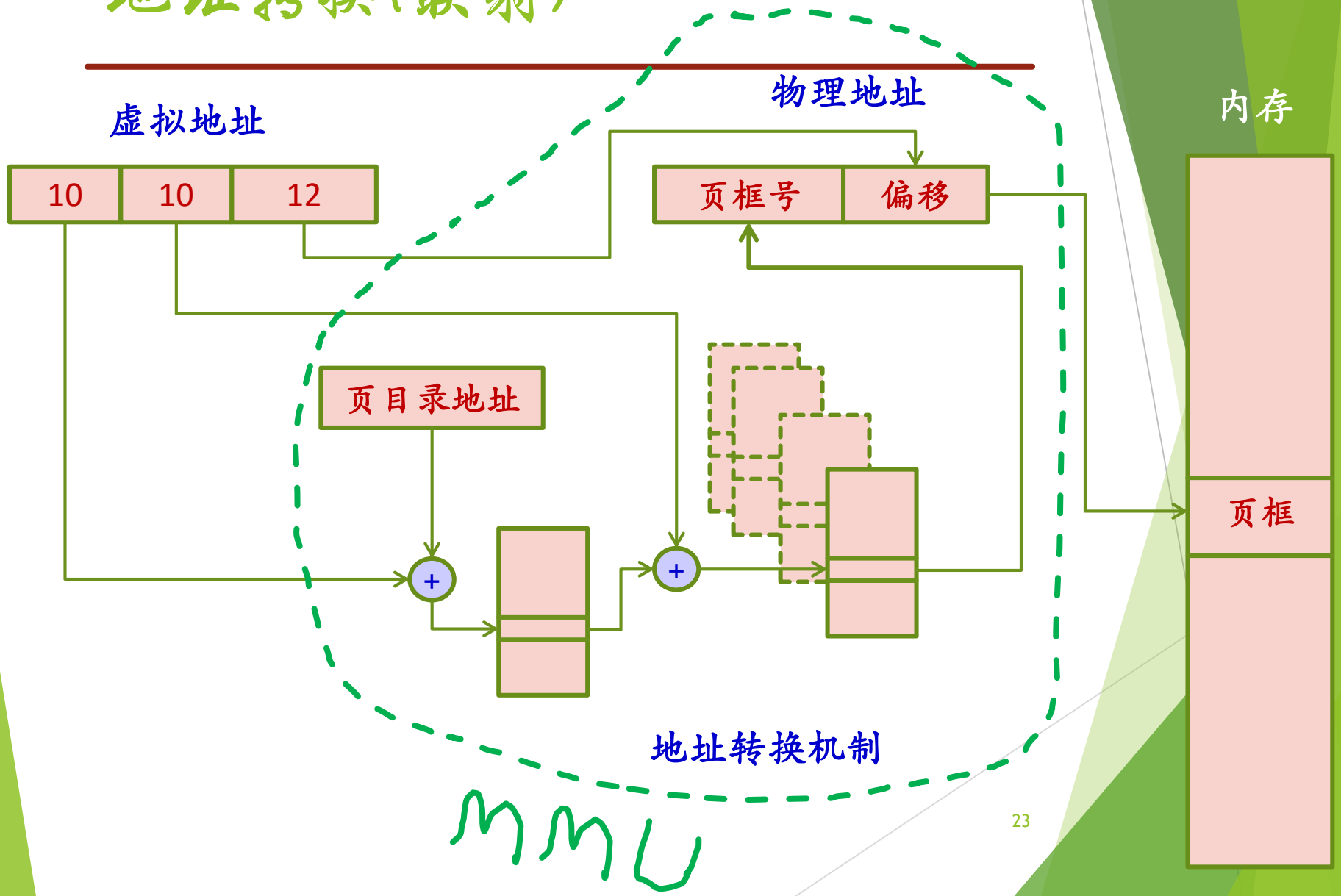
} else {

页框号 = 页表[虚页号]

物理地址 = 页框号 拼接 页内偏移

}

地址转换(映射)



5. 快表(TLB)的引入

问题

- 页表→两次或两次以上的内存访问
- CPU的指令处理速度与内存指令的访问速度差异大，CPU的速度得不到充分利用

如何加快地址映射速度，以改善系统性能？

程序访问的局部性原理→引入快表

快表是什么?

- 相联存储器（associative memory）

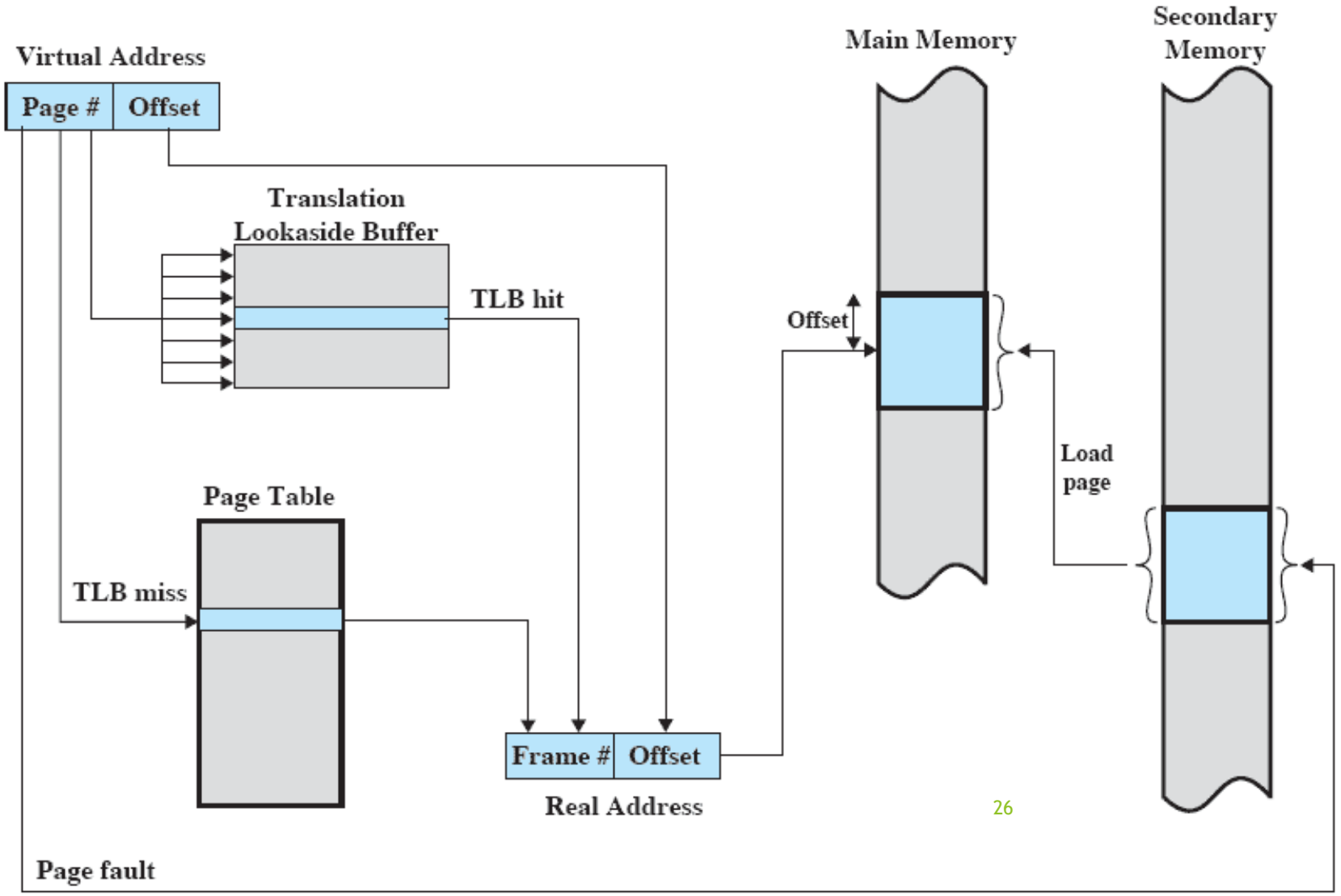
  特点：按内容并行查找
- TLB —— Translation Look-aside Buffers
- 保存正在运行进程的页表的子集(部分表项)

- 快表的作用：加快地址映射速度，改善系统性能
- 工作原理：采用联想映射技术按内容同时查找

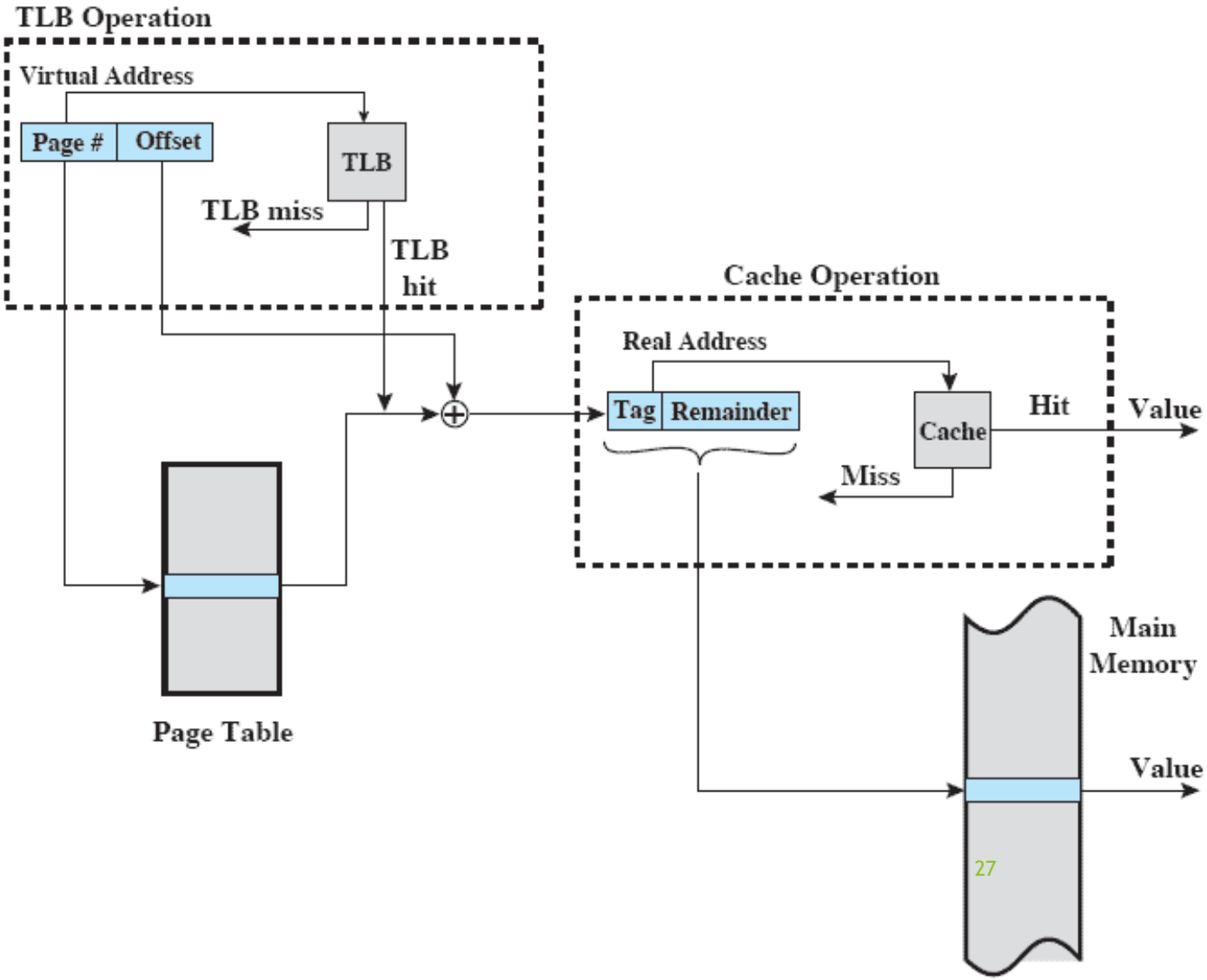
快表的大小、位置

快表的置换问题?

TLB的使用


Virtual Address
Page #
Offset
Translation
Lookaside Buffer
TLB hit
Page Table
TLB miss
Frame #
Offset
Real Address
Main Memory
Secondary
Memory
Offset
Load
page
Page fault

TLB与高速缓存操作


TLB Operation
Virtual Address
Page #
Offset
TLB
TLB miss
TLB hit
Cache Operation
Real Address
Tag
Remainder
Cache
Hit
Value
Miss
Page Table
Main Memory
Value

6.缺页异常处理

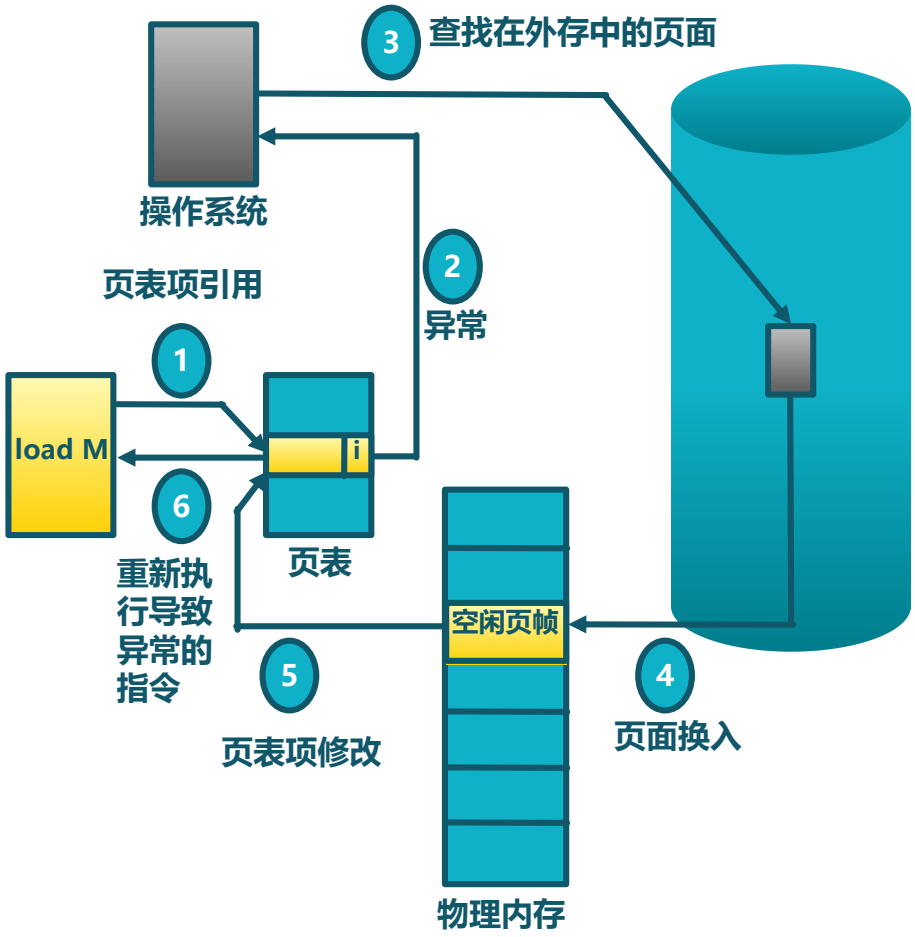
预取一些页面

- 一种Page Fault

  页错误、页面错误、页故障、页失效

- 在地址映射过程中，硬件检查页表时发现所要访问的页不在内存，则产生该异常——缺页异常
- 操作系统执行缺页异常处理程序：获得磁盘地址，启动磁盘，将该页调入内存
  - 如果内存中有空闲页框，则分配一页，将新调入页装入内存，并修改页表中相应页表项的驻留位及相应的页框号
  - 若内存中没有空闲页框，则要置换某一页；若该页在内存期间被修改过，则要将其写回磁盘

缺页异常（缺页中断）的处理流程



A.在内存中有空闲物理页面时，分配一物理页帧f，转第E步；

B.依据页面置换算法选择将被替换的物理页帧f，对应逻辑页q

C.如q被修改过，则把它写回外存；

D.修改q的页表项中驻留位置为0；

E.将需要访问的页p装入到物理页面f

F.修改p的页表项驻留位为1,物理页帧号为f；

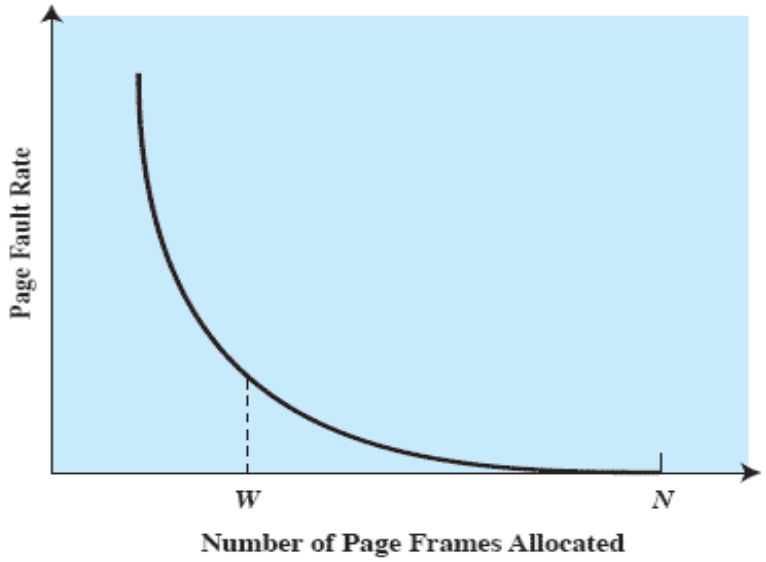
G.重新执行产生缺页的指令

Acknowledgement: 清华大学 向勇、陈渝

7. 驻留集管理

- 驻留集大小

  给每个进程分配多少页框?



- 固定分配策略

  进程创建时确定

  可以根据进程类型（交互、批处理、应用类）或者基于程序员或系统管理员的需要来确定

- 可变分配策略

  根据缺页率评估进程局部性表现

  ✓ 缺页率高→增加页框数

  ✓ 缺页率低→减少页框数

  ✓ 系统开销

基本思想、设计与实现问题、……

页面置换算法

1. 置换问题

- 置换范围

  计划置换页面的集合是局限在产生缺页中断的进程，还是所有进程的页框?

- **局部置换策略**

  仅在产生本次缺页的进程的驻留集中选择

- **全局置换策略**

  将内存中所有未锁定的页框都作为置换的候选

	局部置换	全局置换
固定分配	√	--
可变分配	√	√

1、当一个新进程装入内存时，给它分配一定数目的页框，使用预先分页或请求分页填满这些页框

2．当发生一次缺页中断时，从产生缺页中断的进程的驻留集中选择一页用于置换

3．不断重新评估进程的页框分配情况，增加或减少分配给它的页框，以提高整体性能

置换策略

放置（placement，分配）
置换（replacement，替换、淘汰）

- 决定置换当前内存中的哪一个页框
- 所有策略的目标 → ***置换最近最不可能访问的页***
- 根据局部性原理，最近的访问历史和最近将要访问的模式间存在相关性，因此，大多数策略都***基于过去的行为来预测将来的行为***
- 注意：置换策略设计得越精致、越复杂，实现的软硬件开销就越大
- 约束条件：不能置换被锁定的页框

典型设计思路

页框锁定

特别是正在I/O的内存页面

为什么要锁定页面?

- 采用虚存后:

  开销 → 使程序运行时间变得不确定

- 给每一页框增加一个锁定位
- 通过设置相应的锁定位，不让操作系统将进程使用的页面换出内存，避免产生由交换过程带来的不确定的延迟
- 例如：操作系统核心代码、关键数据结构、I/O缓冲区

Windows中的VirtualLock和VirtualUnLock函数

2. 页面置换(replacement)算法

又称页面淘汰（替换）算法

最优算法→最近未使用→先进先出→第二次机会→时钟算法→最近最少使用→最不经常使用→老化算法→工作集→工作集时钟

理想(最佳、最优)置换算法(OPT)

- 设计思想:

  置换以后不再需要的或最远的将来才会用到的页面

- 实现?
- 作用?

先进先出页面置换算法（FIFO）

- 选择在内存中驻留时间最长的页并置换它

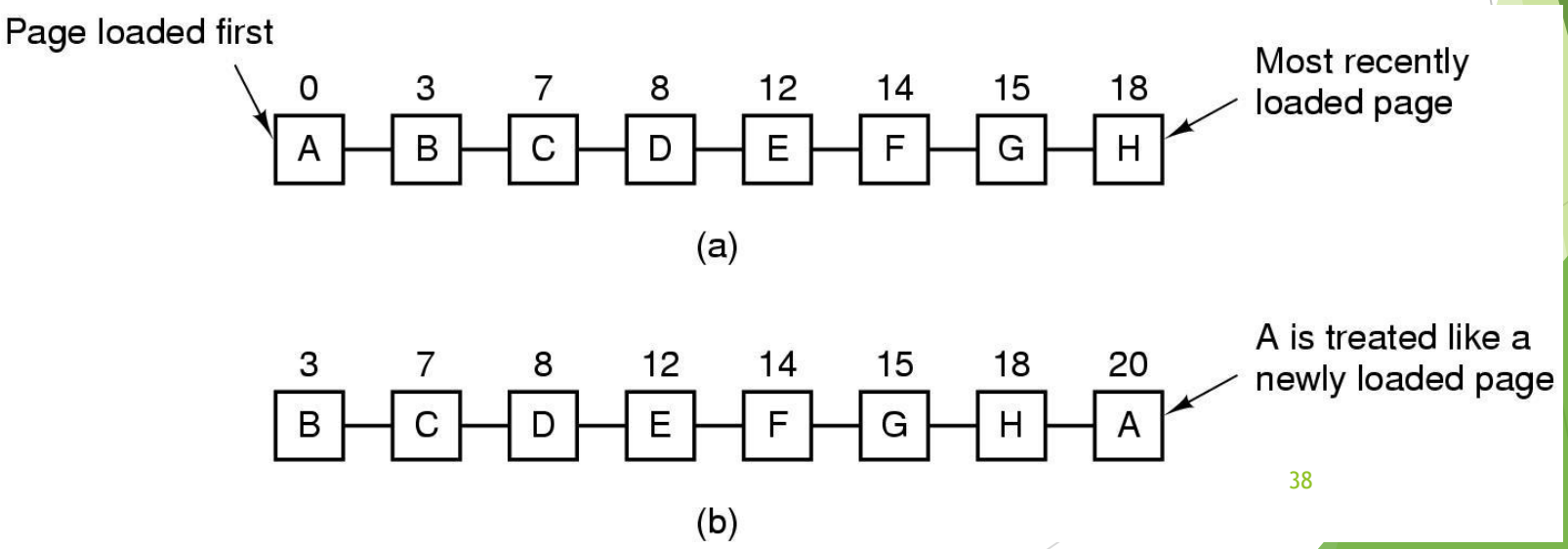
  对照：超市撤换商品

- 实现：页面链表法

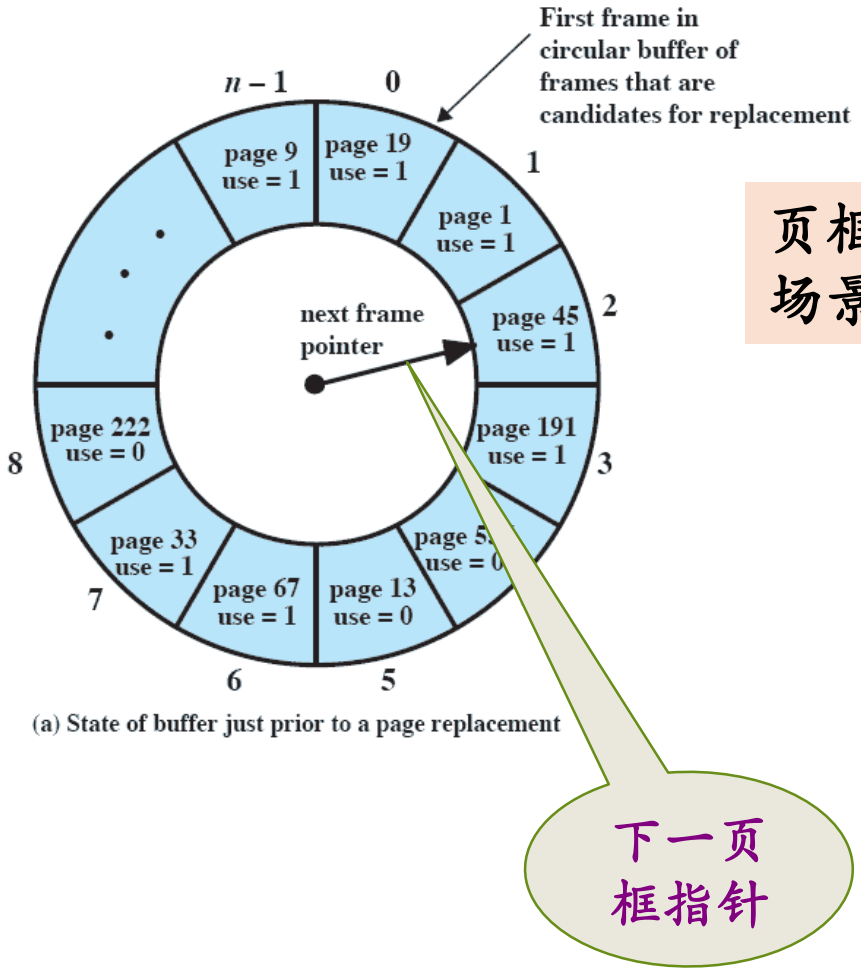
第二次机会置换算法(SCR)

SCR-Second Chance

按照先进先出算法选择某一页面，检查其访问位R，如果为0，则置换该页；如果为1，则给第二次机会，并将访问位置0



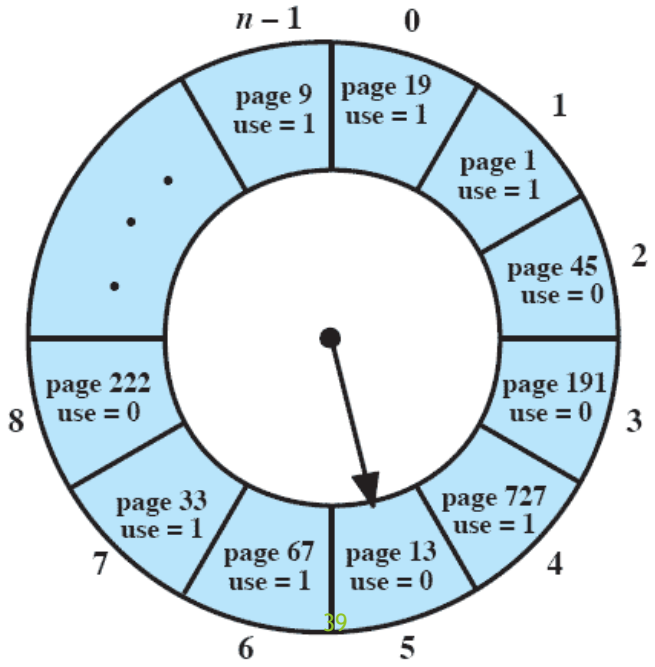
时钟算法(Clock)



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

页框组织成循环缓冲区

场景：页727进入



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

最近未使用算法(NRU)(1/2)

Not Recently Used

选择在最近一段时间内未使用过的一页并置换

实现：设置页表表项的两位

访问位（R），修改位（M）

如果硬件没有这些位，则可用软件模拟（做标记）

启动一个进程时，R、M位置0

R位被定期清零（复位）

最近未使用算法(NRU)(2/2)

发生缺页中断时，操作系统检查R，M:

第0类：无访问，无修改

第1类：无访问，有修改

第2类：有访问，无修改

第3类：有访问，有修改

算法思想:

随机从编号最小的非空类中选择一页置换

NRU的时钟算法实现

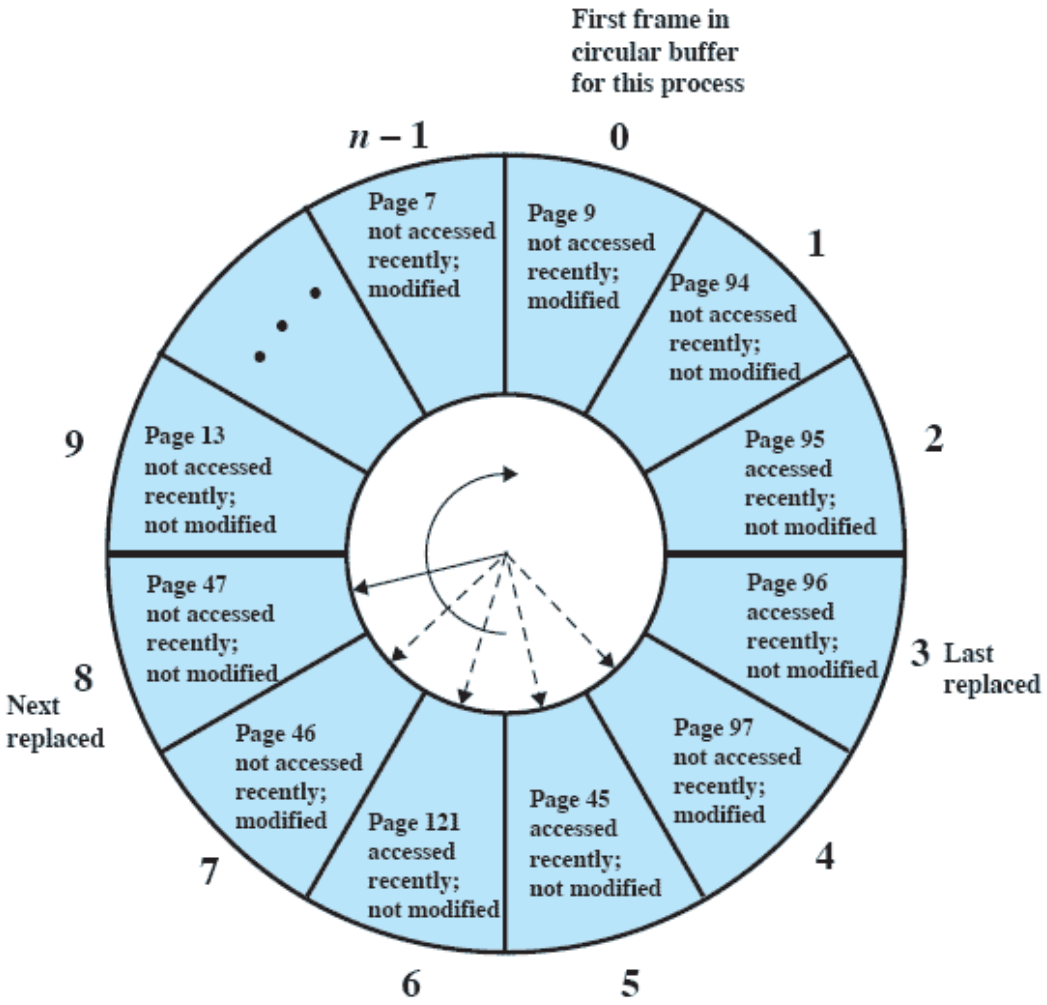
1．从指针的当前位置开始，扫描页框缓冲区，选择遇到的第一个页框（*r*=0；*m*=0）用于置换(本扫描过程中，对访问位不做任何修改)

2．如果第1步失败，则重新扫描，选择第一个（*r*=0；*m*=1）的页框(本次扫描过程中，对每个跳过的页框，将其访问位设置成0)

3．如果第2步失败，指针将回到它的最初位置，并且集合中所有页框的访问位均为0。重复第1步，并且，如果有必要，重复第2步。这样将可以找到供置换的页框

提示：与上面介绍的NRU有区别

时钟算法实现



该策略用于较早版本的macintosh虚拟存储方案

优先选择不需要写回磁盘的页面，节省时间

最近最少使用算法(LRU)

Least Recently Used

选择最后一次访问时间距离当前时间最长的一页并置换

即置换未使用时间最长的一页

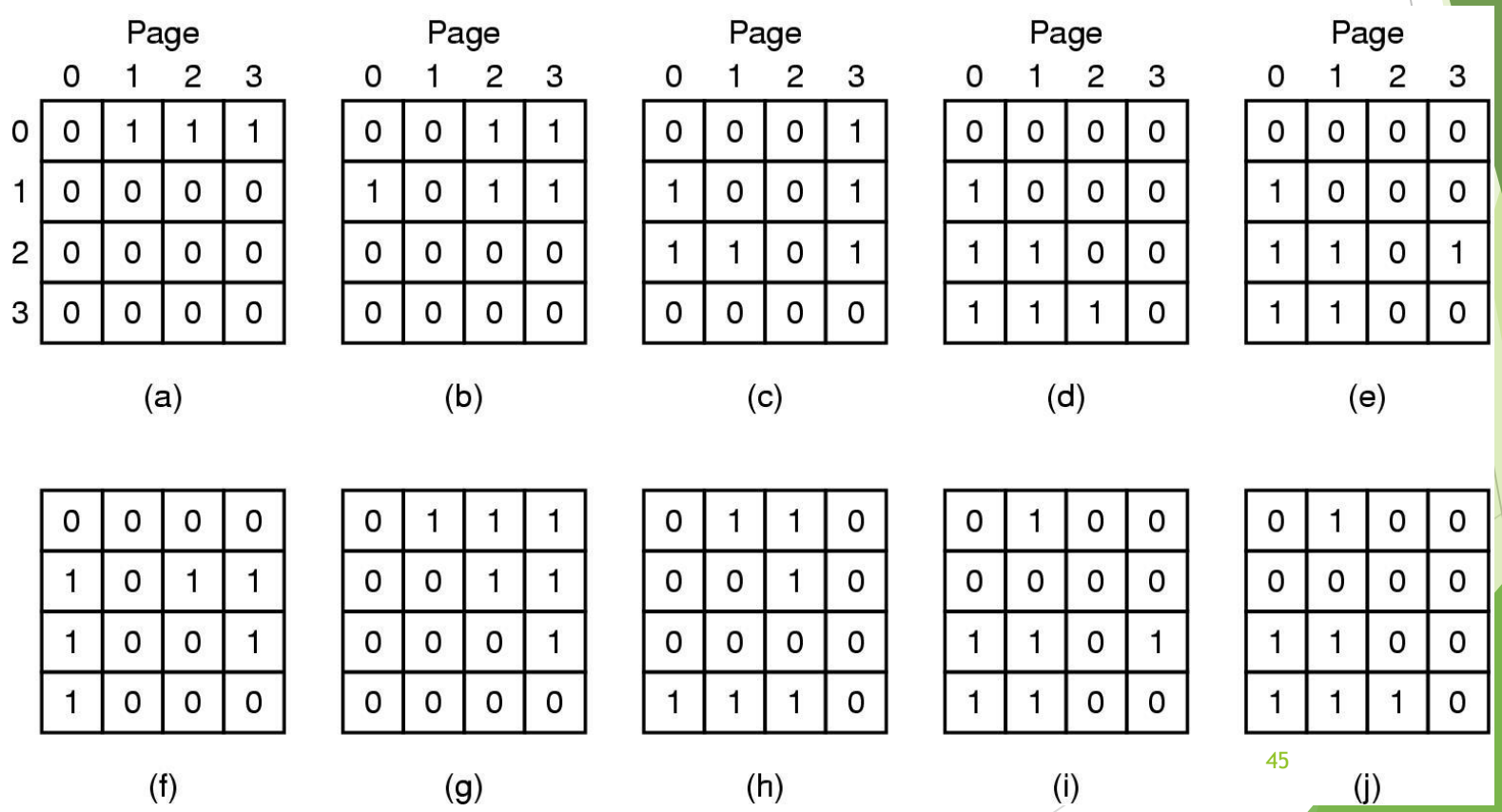
- **性能接近OPT**
- 实现：时间戳 或 维护一个访问页的栈

  → 开销大

LRU算法的一种硬件实现

思考题：解释其理由

- 页面访问顺序0, 1, 2, 3, 2, 1, 0, 3, 2, 3



最不经常使用算法(NFU)

Not Frequently Used

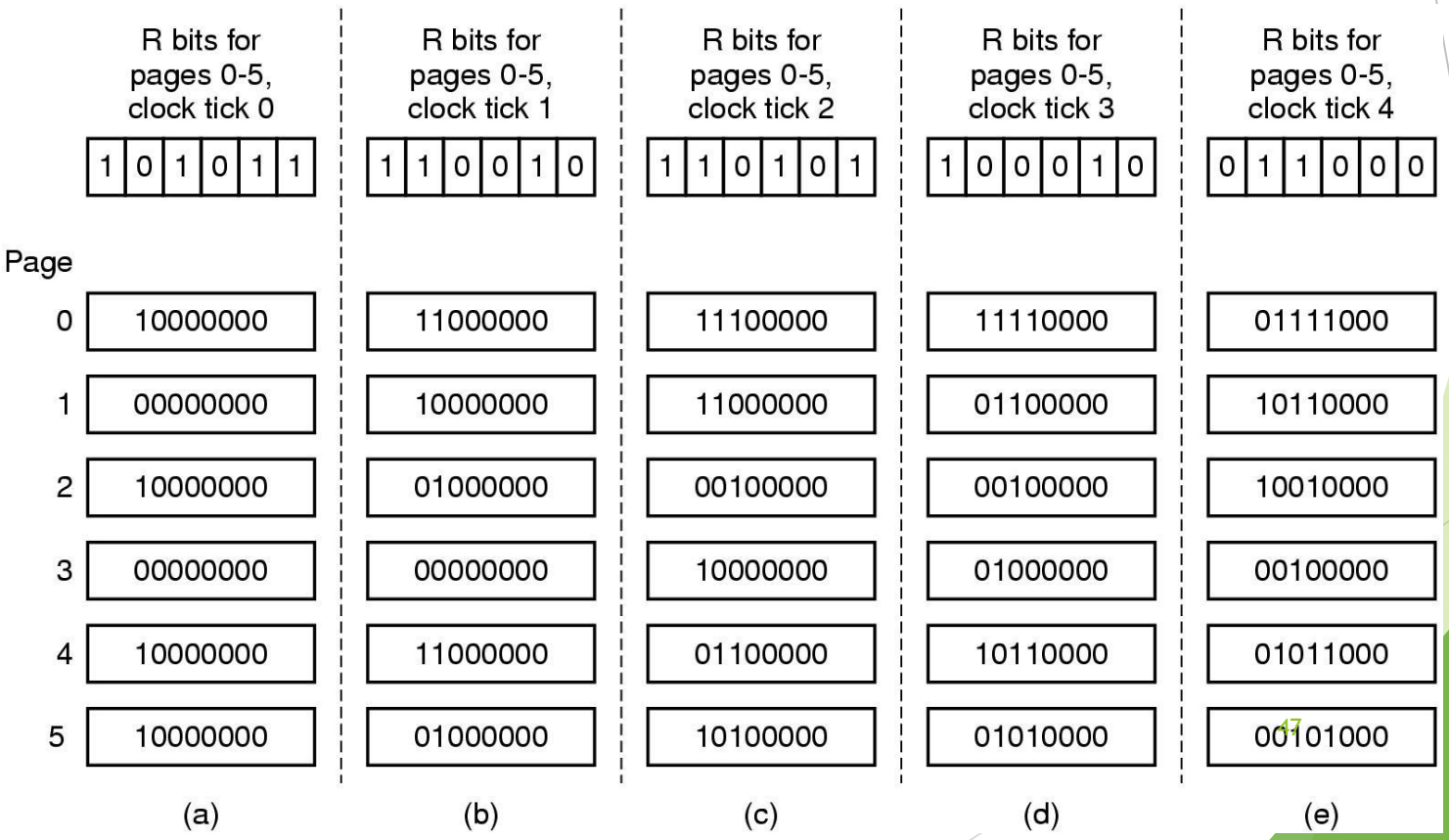
选择访问次数最少的页面置换

- LRU的一种软件解决方案
- 实现：
  - 软件计数器，一页一个，初值为0
  - 每次时钟中断时，计数器加R
  - 发生缺页中断时，选择计数器值最小的一页置换

思考题：与LRU的比较

老化算法(Aging)

改进（模拟LRU）：计数器在加R前先右移一位
R位加到计数器的最左端



3.页面置换算法应用

例子：系统给某进程分配3个页框(固定分配策略)，初始为空

进程执行时，页面访问顺序为：

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

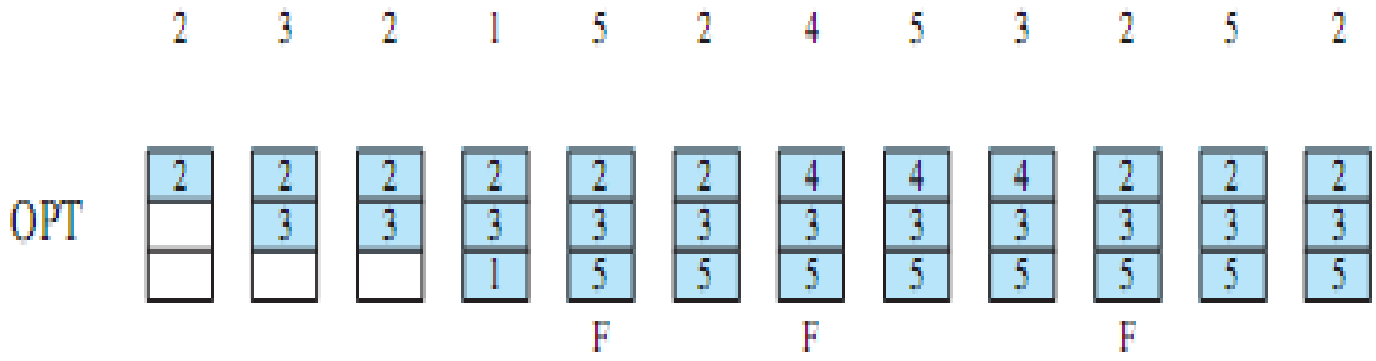
要求：计算应用FIFO、LRU 、OPT算法时的缺页次数

应用FIFO、LRU页面置换算法

	2	3	2	1	5	2	4	5	3	2	5	2
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F

	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		

应用OPT页面置换算法



4. Belady现象

例子：系统给某进程分配m个页框，初始为空

页面访问顺序为

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

采用FIFO算法，计算当 m=3 和 m=4 时的缺页中断次数

m=3时，缺页中断9次；m=4时，缺页中断10次

注：FIFO页面置换算法会产生异常现象（Belady现象），即：当分配给进程的物理页面数增加时，缺页次数反而增加

5.影响缺页次数的因素

- 页面置换算法*
- 页面本身的大小√
- 程序的编制方法√
- 分配给进程的物理页面数√

颠簸（Thrashing，抖动）

虚存中，页面在内存与磁盘之间频繁调度，使得调度页面所需的时间比进程实际运行的时间还多，这样导致系统效率急剧下降，这种现象称为颠簸或抖动

页面尺寸问题

- 确定页面大小对于分页的硬件设计非常重要而对于操作系统是个可选的参数

小页面?
大页面?

最优页面大小
$P=\sqrt{2se}$

- 要考虑的因素:
  - 内部碎片
  - 页表长度
  - 辅存的物理特性

- Intel 80x86/Pentium: 4096 或 4M

- 多种页面尺寸: 为有效使用TLB带来灵活性，但给操作系统带来复杂性

程序编制方法对缺页次数的影响

例子：系统给某进程分配了一个页框；页面大小4K；
矩阵A[1024][1024]；按行存放

程序编制方法1：

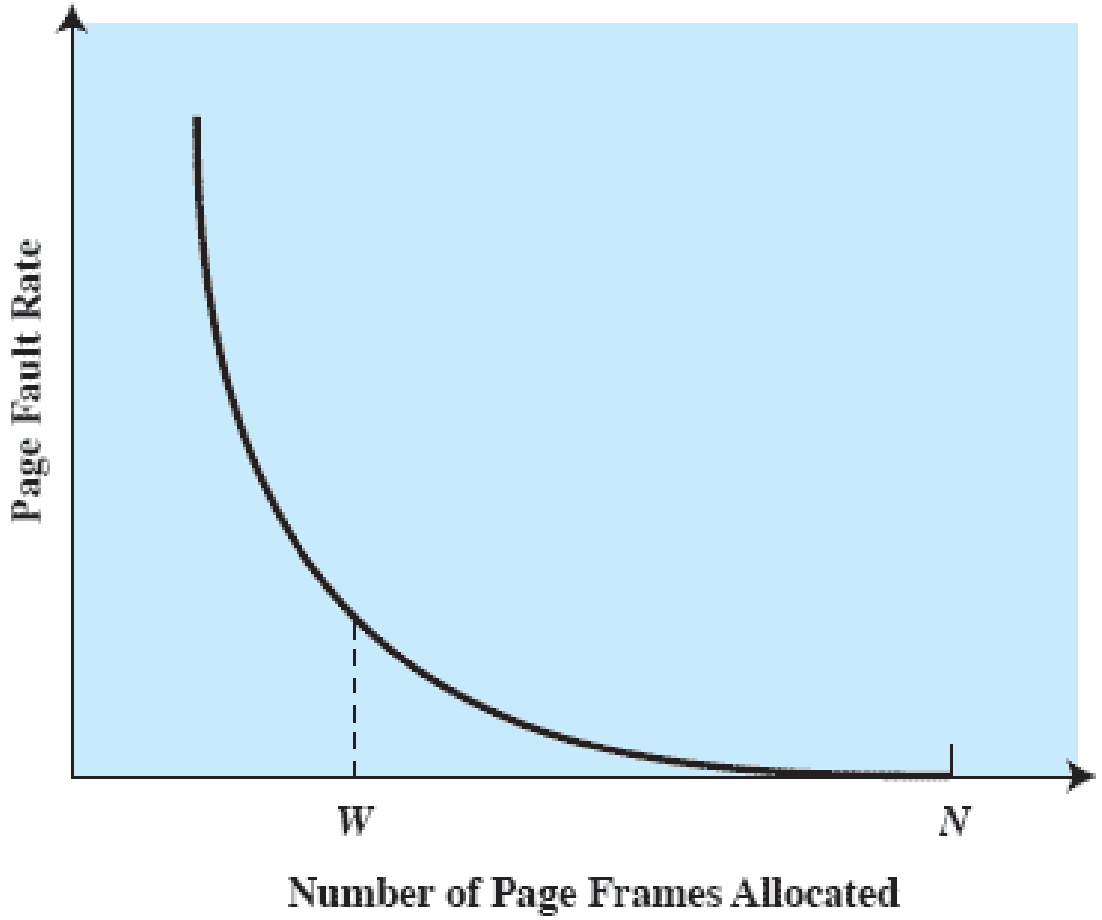
```
for (j = 0; j < 1024; j++)
   for (i = 0; i < 1024; i++)
      A[i][j] = 0;
```

程序编制方法2：

```
for (i=0; i<1024; i++)
   for (j=0; j<1024; j++)
      A[i][j] = 0;
```

缺页
几次？

分配给进程的页框数与缺页率的关系



6. 工作集(Working Set)模型(1/3)

基本思想:

根据程序的局部性原理，一般情况下，进程在一段时间内总是集中访问一些页面，这些页面称为活跃页面，如果分配给一个进程的页框太少了，使该进程所需的活跃页面不能全部装入内存，则进程在运行过程中将频繁发生中断

如果能为进程提供与活跃页面数相等的页框数，则可减少缺页中断次数

由Denning提出(1968)

工作集(Working Set)模型(2/3)

工作集$W(t, \Delta)$

= 该进程在过去的Δ个虚拟时间单位中使用的虚拟页面集合

工作集内容取决于三个因素：

- 访页序列特性
- 当前时刻t
- 工作集窗口长度(Δ)

窗口越大，工作集就越大

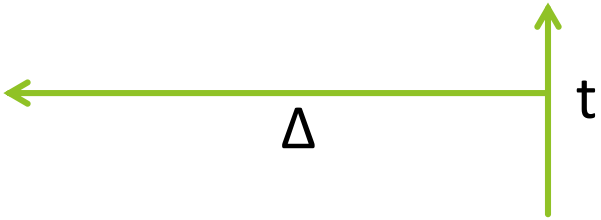
工作集(Working Set)模型(3/3)

例子:

2615777516234123444343444 1327

$W(t_1,10)=\{1,2,5,6,7\}$ $W(t_2,10)=\{3,4\}$

2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 7



W(t, Δ) = {1, 2, 5, 6, 7} {3, 4} {3, 4} {1, 2, 3, 4} {1, 3, 4, 7}

工作集与驻留集

* 驻留集：当前时刻，进程实际驻留在内存当中的页框集合

两者的关系：

* 工作集是进程在运行过程中固有的性质
* 驻留集取决于系统分配给进程的页框数和页面置换算法

* 监视每个进程的工作集
* 周期性地从一个进程的驻留集中移去那些不在它的工作集中的页（可使用LRU策略 ）
* 只有当一个进程的工作集在内存中时，才可以更好地执行该进程，即进程的驻留集包括了它的工作集）

工作集算法(1/3)

基本思路:

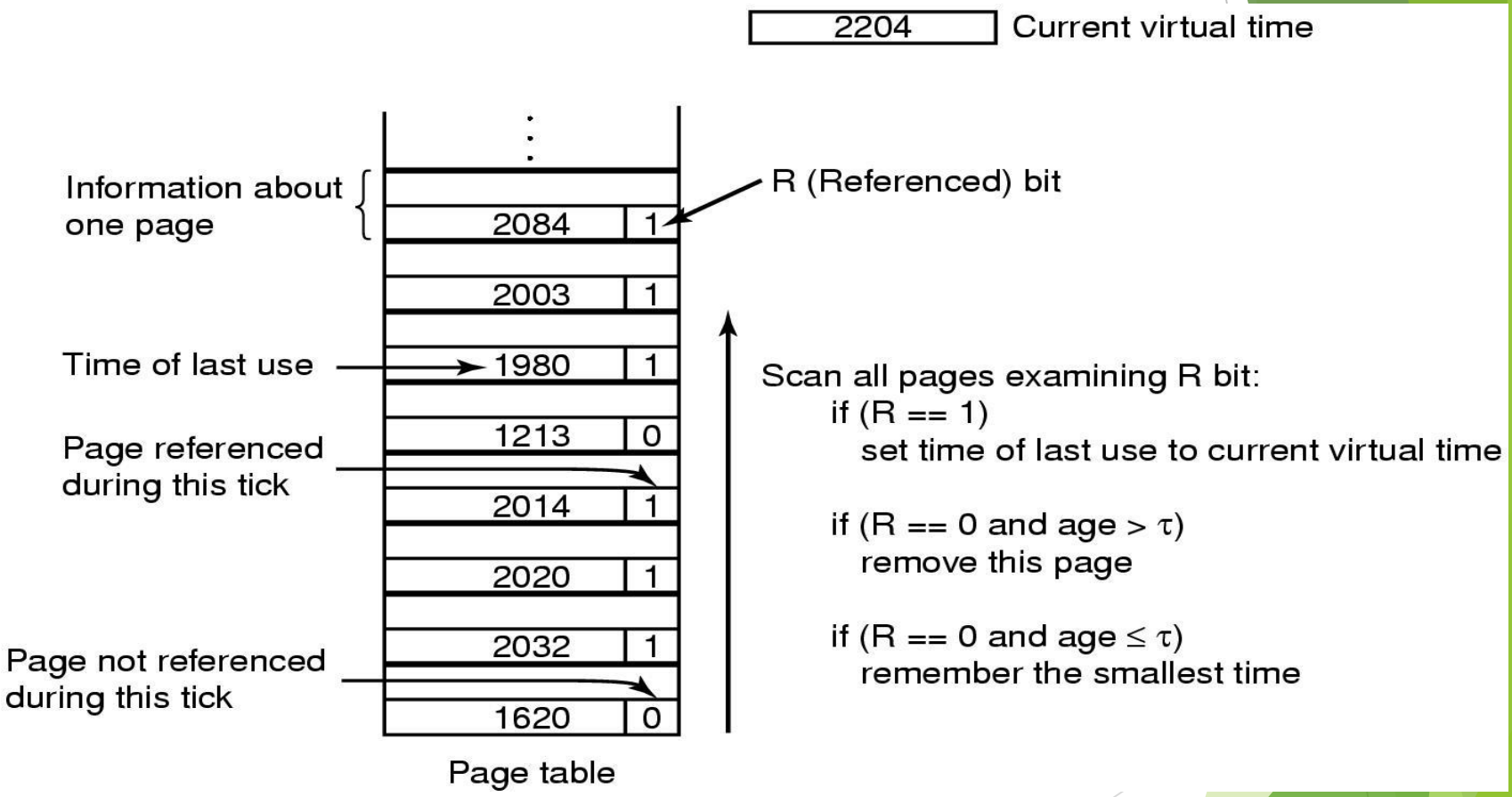
找出一个不在工作集中的页面并置换它

思路:

- 每个页表项中有一个字段:记录该页面最后一次被访问的时间
- 设置一个时间值T
- 判断:

  根据一个页面的访问时间是否落在“当前时间-T”之前或之中决定其在工作集之外还是之内

工作集算法(2/3)


2204
Current virtual time
Information about one page
R (Referenced) bit
2084
1
2003
1
Time of last use
1980
1
1213
0
Page referenced during this tick
2014
1
2020
1
2032
1
Page not referenced during this tick
1620
0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

工作集算法(3/3)

实现:

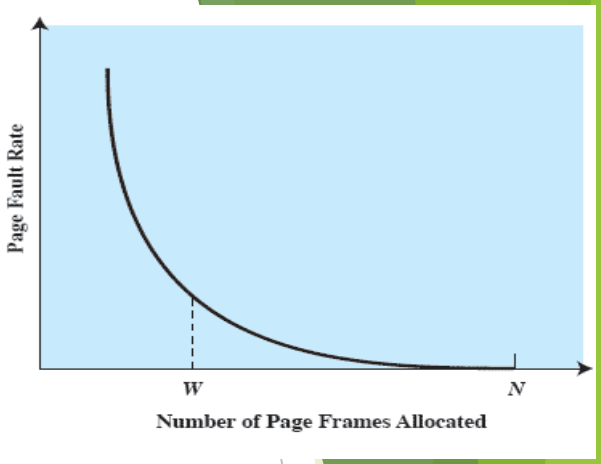
扫描所有页表项，执行操作

1. 如果一个页面的R位是1，则将该页面的最后一次访问时间设为当前时间，将R位清零
2. 如果一个页面的R位是0，则检查该页面的访问时间是否在“当前时间-T”之前

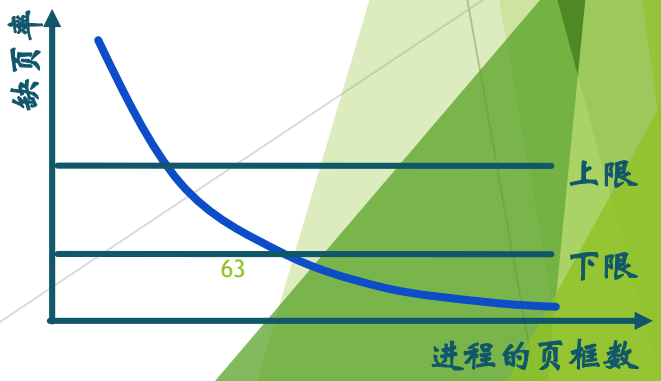
(1) 如果是，则该页面为被置换的页面;

(2) 如果不是，记录当前所有被扫描过页面的最后访问时间里面的最小值。扫描下一个页面并重复1、2

工作集算法讨论



- 工作集策略的思想是有效的
- 许多操作系统都试图采用近似工作集策略
- 其中的一种方法是考虑进程的缺页率，通过监视缺页率达到类似的结果——缺页率算法
- 当增大一个进程的驻留集时，缺页率会下降，工作集的大小会降到图中W点所标记的位置
- 方法：设置一个进程的缺页率的最小阈值和最大阈值，根据此增减驻留集大小



页面置换算法小结

算法	评价
OPT	不可实现，但可作为基准
NRU	LRU的很粗略的近似
FIFO	可能淘汰重要的页面
Second Chance	比FIFO有很大的改善
Clock	现实的
LRU	很优秀，但很难实现
NFU	LRU的相对粗略的近似
Aging	非常近似LRU的有效算法
Working set	实现起来开销很大
WSClock	好的有效的算法

7. 清除策略(1/2)

- 分页系统工作的最佳状态：发生缺页异常时，系统中有大量的空闲页框
- *结论：保存一定数目的页框供给比使用所有内存并在需要时搜索一个页框有更好的性能*

- 设计一个分页守护进程（paging daemon），多数时间处于睡眠状态，可定期唤醒以检查内存状态
- 如果空闲页框过少，分页守护进程通过预设的页面置换算法选择页面换出内存
- 如果页面装入内存后被修改过，则将它们写回磁盘

分页守护进程可保证所有的空闲页框是“干净”的

清除策略(2/2)

- 当需要使用一个已置换出的页框时，如果该页框还没有被新内容覆盖，则将它从空闲页框缓冲池中移出即可恢复该页面

清除策略实现(A.S.Tanenbaum)

- 使用一个双指针时钟，前指针由分页守护进程控制：当它指向一个“脏”页面时，就把该页面写回磁盘，前指针向前移动；当它指向一个干净页面时，仅仅向前移动指针
- ✓ 后指针用于页面置换，与标准时钟算法一样
- ✓ 由于分页守护进程的工作，后指针命中干净页面的概率会增加

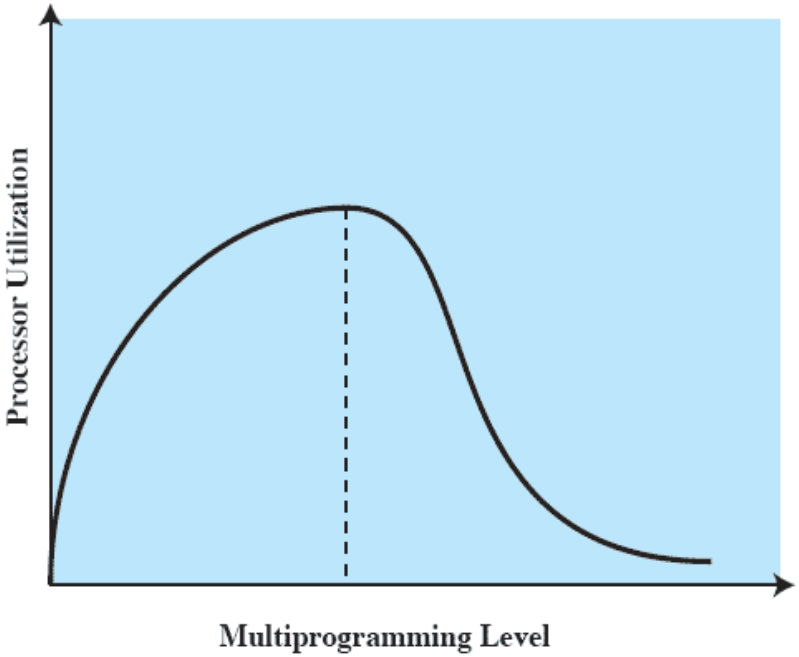
页缓冲技术

- 目的：提高性能
- 思路
  - 不丢弃置换出的页，将它们放入两个表之一：如果未被修改，则放到*空闲页链表*中，如果修改了，则放到*修改页链表*中
  - 被修改的页以簇方式写回磁盘(不是一次只写一个，减少I/O操作的数量，从而减少了磁盘访问时间)
  - 被置换的页仍然保留在内存中，一旦进程又要访问该页，可以迅速将它加入该进程的驻留集合(代价很小)

8.加载控制

系统并发度：驻留在内存中的进程数目

通过调节并发进程数进行系统负载控制



解决方案：进程挂起

即释放一部分进程所占有的页面

→将它们交换到磁盘上

问题：

选择哪些进程？

内存映射文件、策略和机制分离、……

其他设计及实现问题

内存映射文件

- 基本思想

进程通过一个系统调用（例如mmap）将一个文件映射到其虚拟地址空间的一个区域，访问这个文件就象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

内存映射——mmap()函数

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

- 将指定文件fd中偏移量offset开始的长度为length个字节的一块信息映射到虚拟空间中起始地址为start、长度为length个字节的一块区域
- 得到vm_area_struct结构的信息，并生成相应页表项，建立文件地址和区域之间的映射关系

prot 指定该区域内页面的访问权限位，对应vm_area_struct结构中的vm_prot字段

PROT_EXE：页面内容由指令组成
PROT_READ：区域内页面可读
PROT_WRITE：区域内页面可写
PROT_NONE：区域内页面不能被访问

内存映射——mmap()函数（续1）

flags指定所映射的对象的类型，对应vm_area_struct结构中的vm_flags字段

MAP_PRIVATE：

私有的写时拷贝对象，对应可执行文件中只读代码区域（.init、.text、.rodata）和已初始化数据区域（.data）

MAP_SHARED：

共享对象，对应共享库文件中的信息

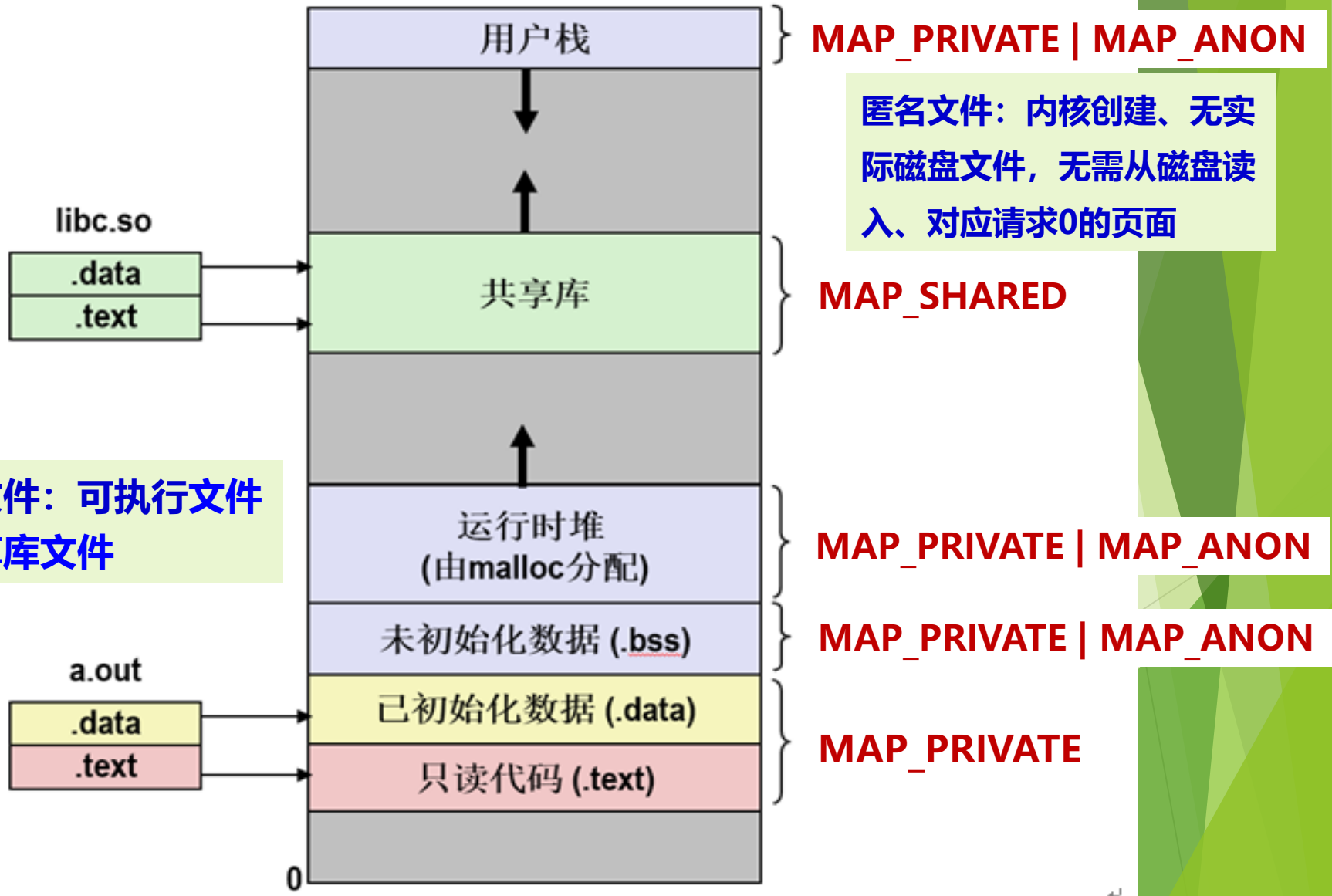
MAP_ANON：

请求0的页，对应内核创建的匿名文件，相应页框用0覆盖并驻留内存

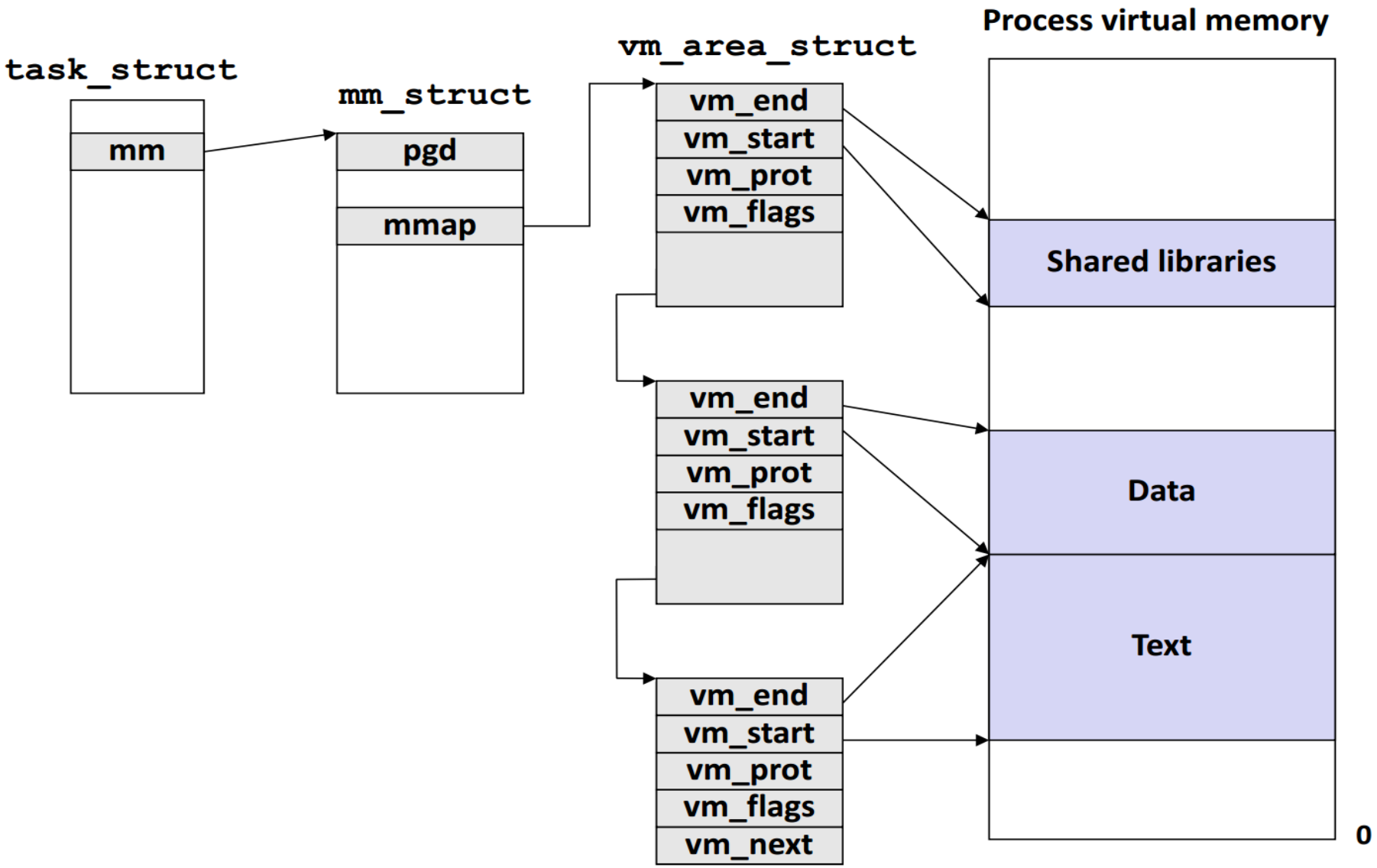
MAP_PRIVATE | MAP_ANON：

未初始化数据（.bss）、堆和用户栈等对应区域

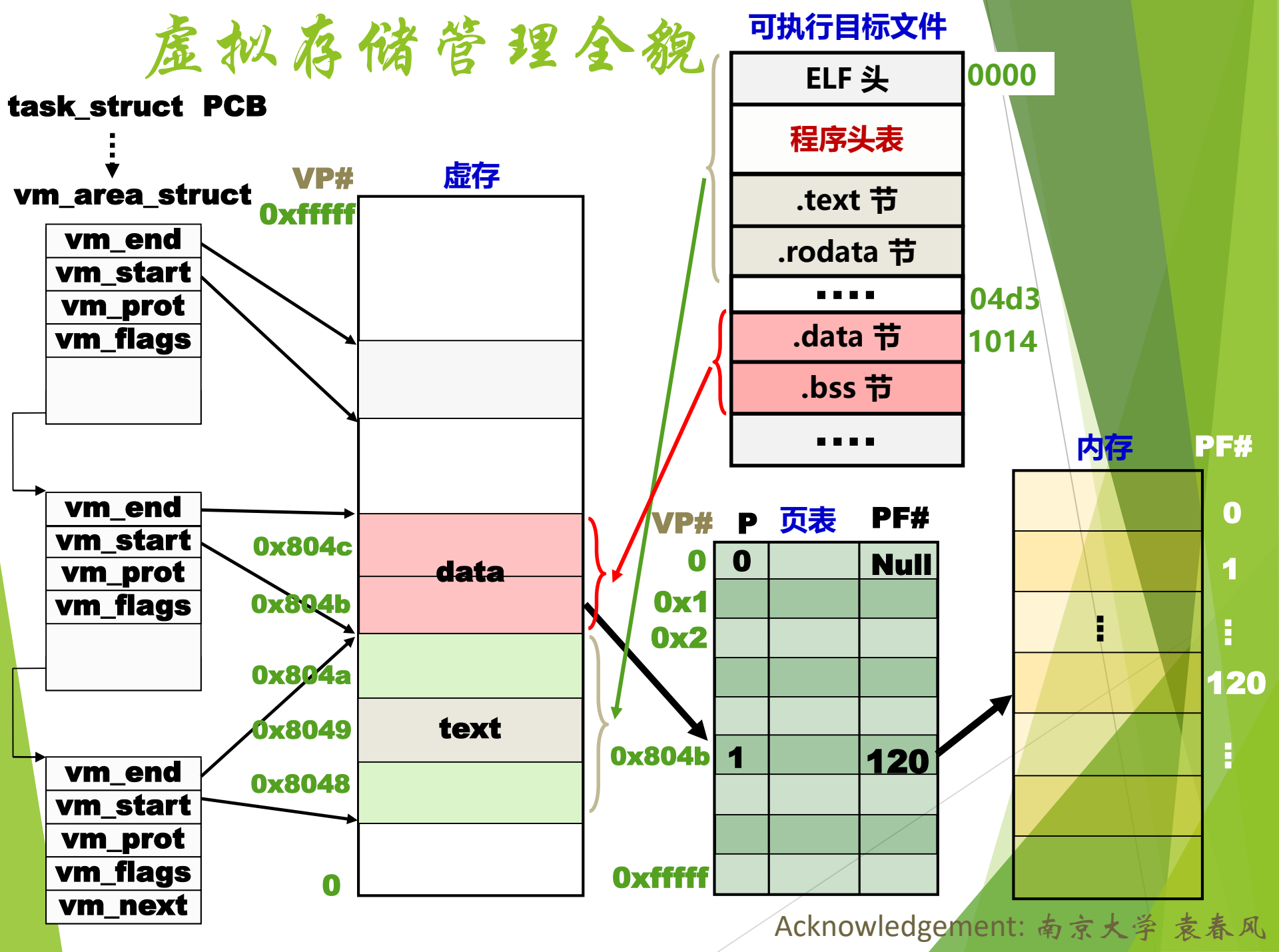
内存映射——mmap()函数（续2）



回顾：ICS课程


task_struct
mm
mm_struct
pgd
mmap
vm_area_struct
vm_end
vm_start
vm_prot
vm_flags
vm_end
vm_start
vm_prot
vm_flags
vm_end
vm_start
vm_prot
vm_flags
vm_next
Process virtual memory
Shared libraries
Data
Text
0

虚拟存储管理全貌



Acknowledgement: 南京大学 袁春风

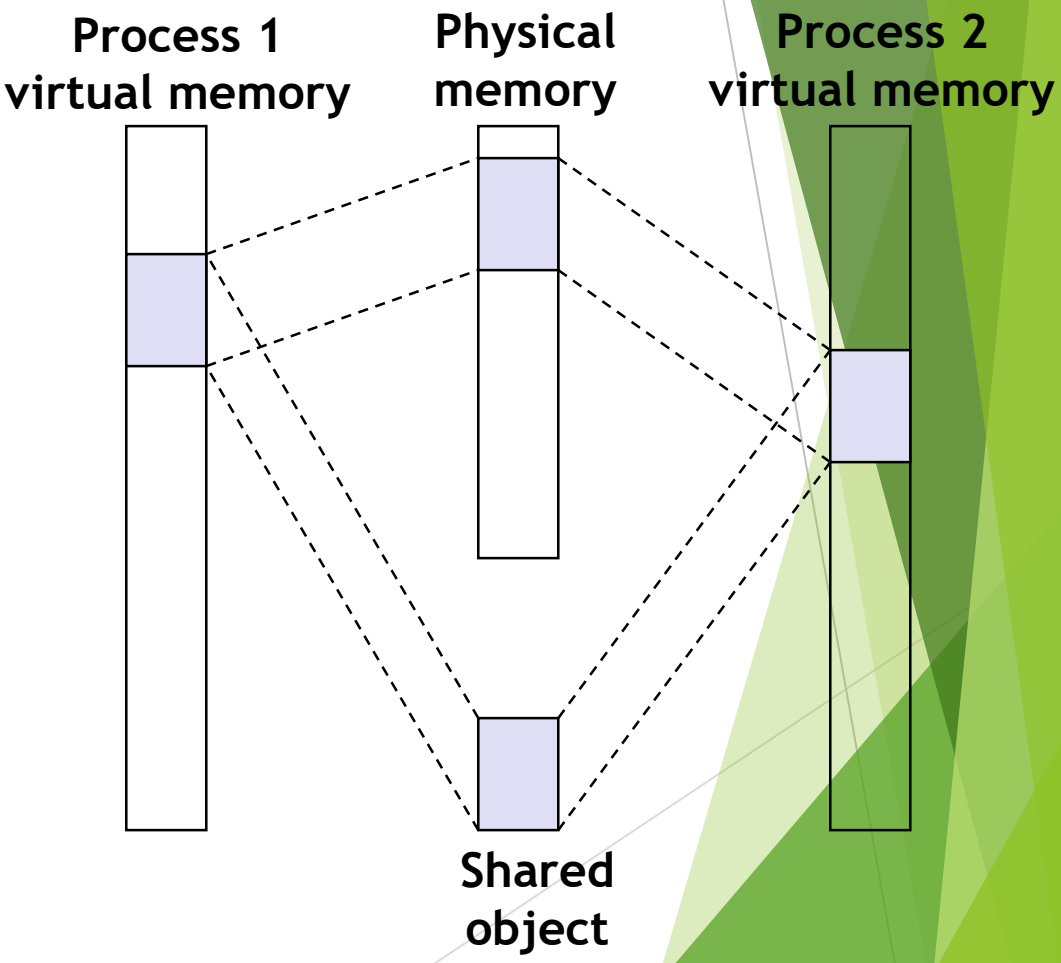
共享库文件中的共享对象

多个进程调用共享库文件中的代码，但共享库代码在内存和硬盘都只需要一个副本

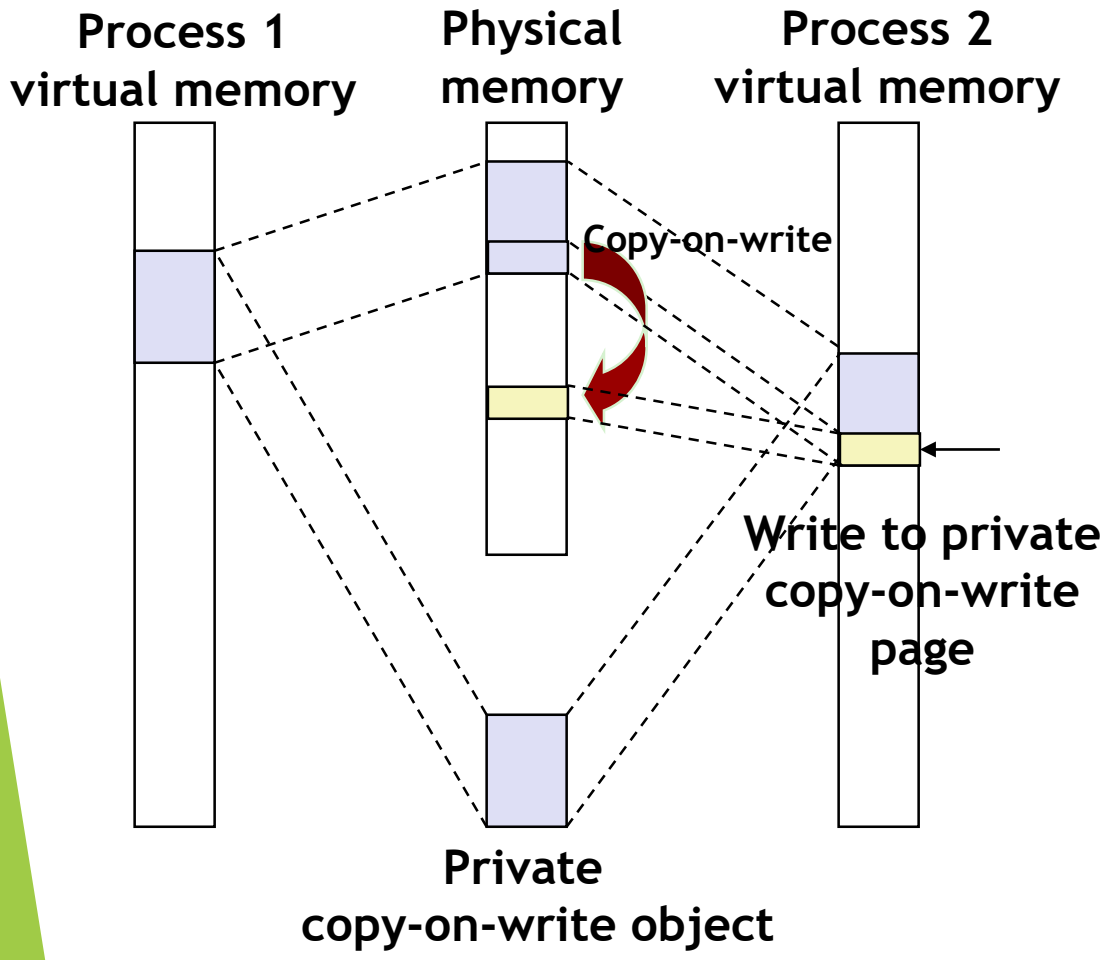
进程1运行过程中，内核为共享对象分配若干页框

进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上即可

一个进程对共享区域进行的写操作结果，对于所有共享同一个共享对象的进程都是可见的，而且结果也会反映在硬盘上对应的共享对象中



私有的写时拷贝对象



同一个可执行文件对应不同进程时，只读代码区一样，可读可写数据区开始也一样，但属于私有对象

为节省主存，多采用写时拷贝技术

进程1运行过程中，内核为对象分配若干页框，并标记为只读

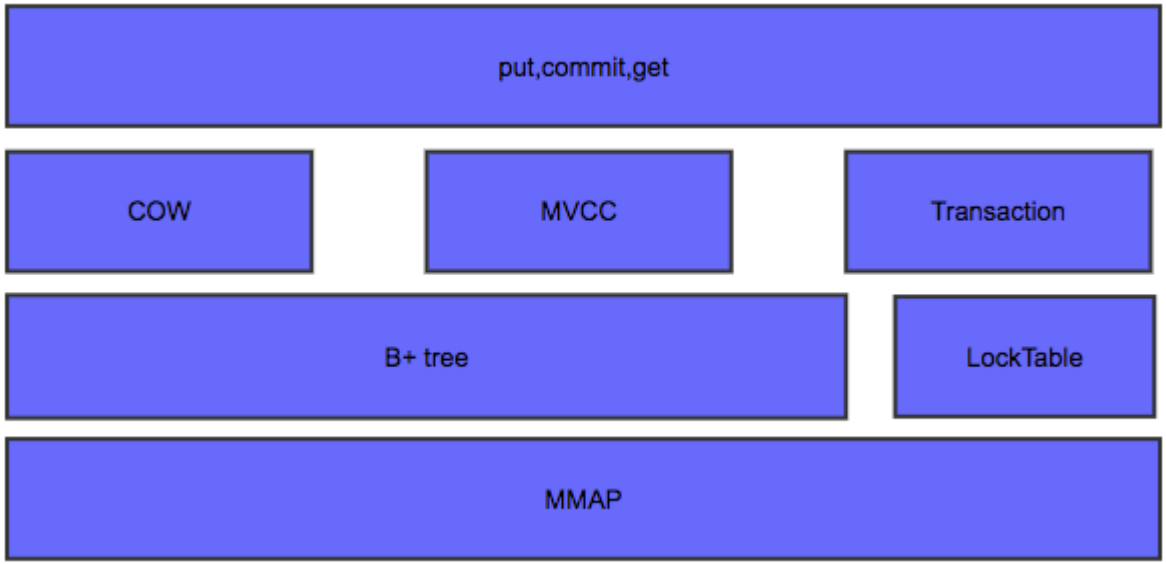
进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上，并标记为只读

若两个进程都只是读或执行，则在内存只有一个副本，节省主存；

若进程2进行写操作，则发生访问违例，此时，内核判断异常原因是进程试图写私有的写时拷贝页，就会分配一个新页框，把内容拷贝到新页框，并修改进程2的页表项

内存映射文件的应用

- LMDB （Lightning Memory-Mapped Database）
- 是内存映射型数据库，通过使用内存映射文件，可以提供更好的输入/输出性能，对于用于神经网络的大型数据集(比如 ImageNet)，可以将其存储在 LMDB 中



策略与机制分离(1/2)

- 控制系统复杂度的重要方法

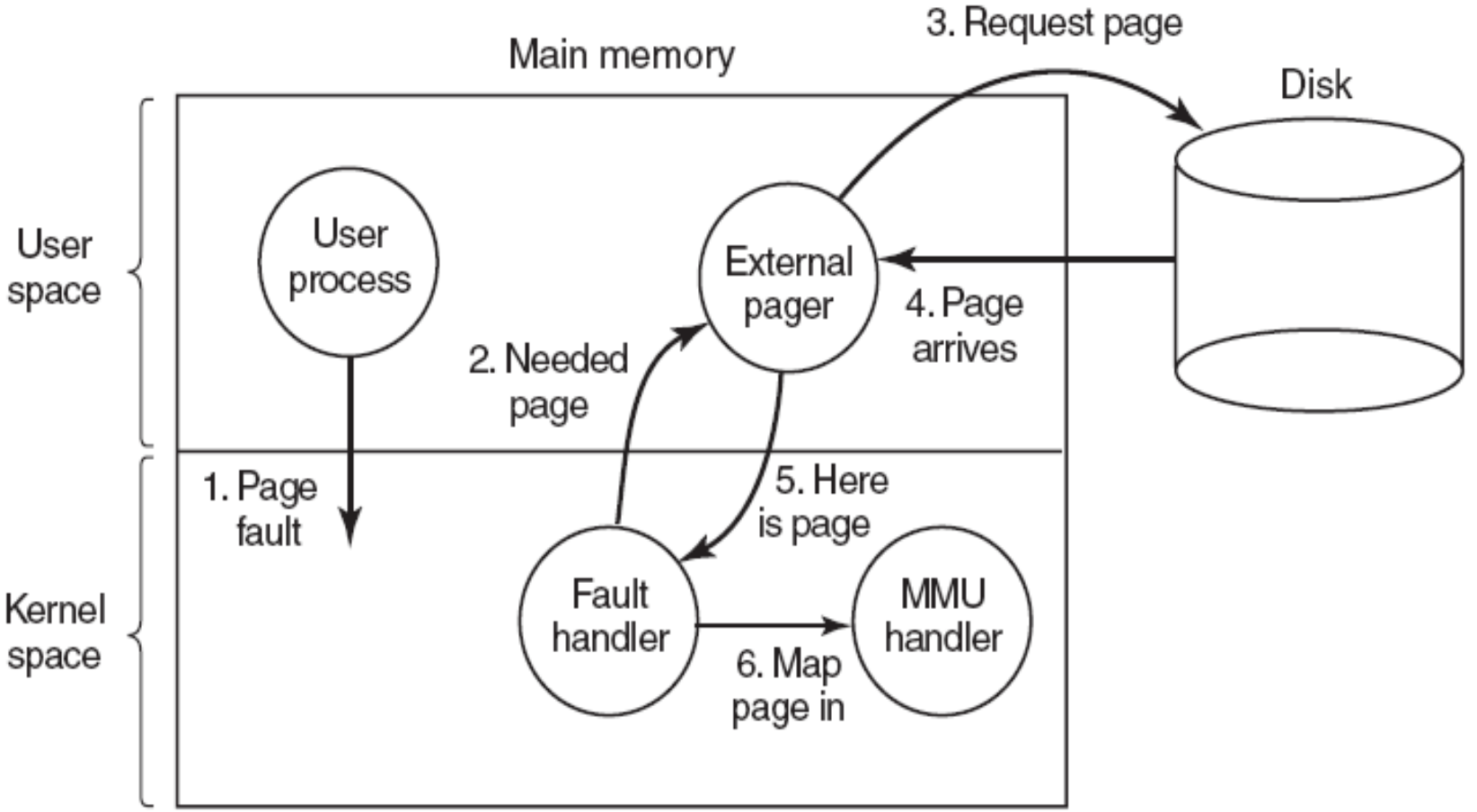
  → 把策略从机制中分离出来

- 讨论（基于Mach）

  存储管理系统被分为三个部分:

  - 底层MMU处理程序(与机器相关)
  - 作为内核一部分的缺页中断处理程序(与机器无关)
  - 运行在用户空间中的外部页面调度程序(策略)

策略与机制分离(2/2)


Main memory
3. Request page
Disk
User space
User process
External pager
4. Page arrives
2. Needed page
1. Page fault
5. Here is page
Kernel space
Fault handler
MMU handler
6. Map page in

重点小结

- 基本概念
  - 虚拟内存、虚拟地址、虚拟地址空间
  - 页表表项、多级页表、倒排页表
  - 快表TLB
- 虚拟内存管理
  - 取页策略（按需、预取）
  - 驻留集管理（固定、可变）
  - 置换范围与策略（局部、全局）
  - 清除策略（页缓冲）
  - 加载控制
- 内存映射文件

作业7

作业提交时间：
2020年12月6日 晚23:30

1、内存管理单元（MMU）、快表（TLB）、页面错误异常（Page Fault）、页表（多级页表、反转页表）、页表项（PTE）、地址保护、放置策略（Placement）、清除策略、置换策略（Replacement）、预取策略、驻留集/工作集。请总结虚拟页式管理机制，要求涵盖上述所有关键词。

2、某64位计算机，虚拟地址空间为2^{48}，页面大小4K，物理内存大小4G。分析对比：采用基于Hash表的反转页表比普通页表节省了多少空间？

作业7

作业提交时间：
2020年12月13日 晚23:30

3、考虑一个进程的页访问序列，工作集为M页框，最初都是空的。页访问串的长度为P，包含N个不同的页号。对任何一种页面置换算法：

a．缺页中断次数的下限是多少？

b．缺页中断次数的上限是多少？

4、在论述一种页面置换算法时，有位作者用在循环轨道上来回移动的雪犁机来模拟说明：雪均匀地落在轨道上，雪犁机以恒定的速度在轨道上不断地循环，轨道上被扫落的雪从系统中消失。

a．哪种页面置换算法可以它来模拟？

b．这一模拟说明了页面置换算法的哪些行为？

作业7

5、阅读（但不限于）以下文章：

① https://blog.csdn.net/joejames/article/details/37958017 Linux内存映射mmap原理分析

② https://www.jianshu.com/p/eece39beee20 深入剖析mmap原理 - 从三个关键问题说起

简要你对讨论mmap的理解。

Thanks

The End