哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

操作系统

Operating Systems

夏文 副教授

xiawen@hit.edu.cn

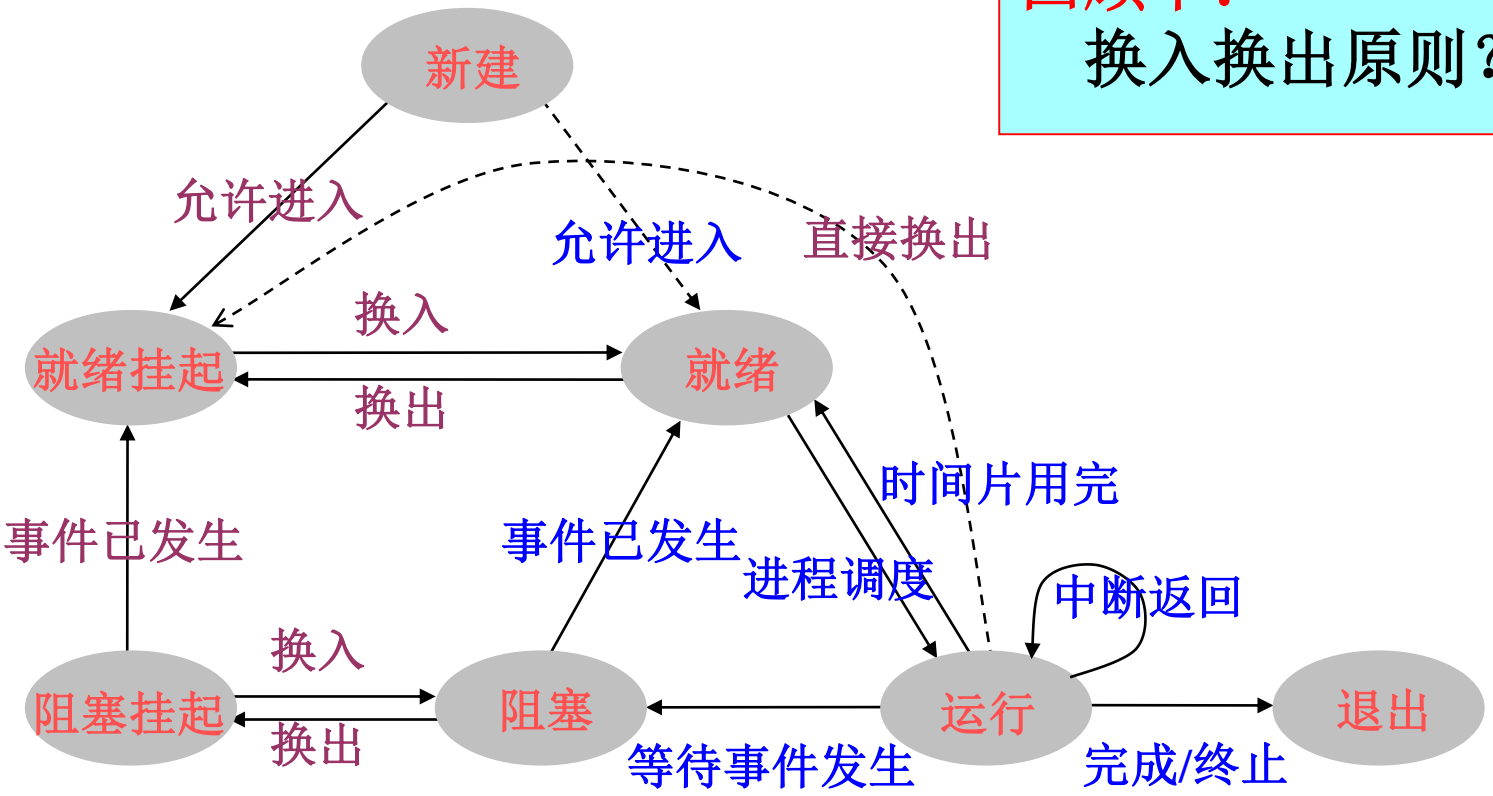
哈尔滨工业大学（深圳）

2019年11月

回顾：进程的描述与表达

进程七状态变迁图

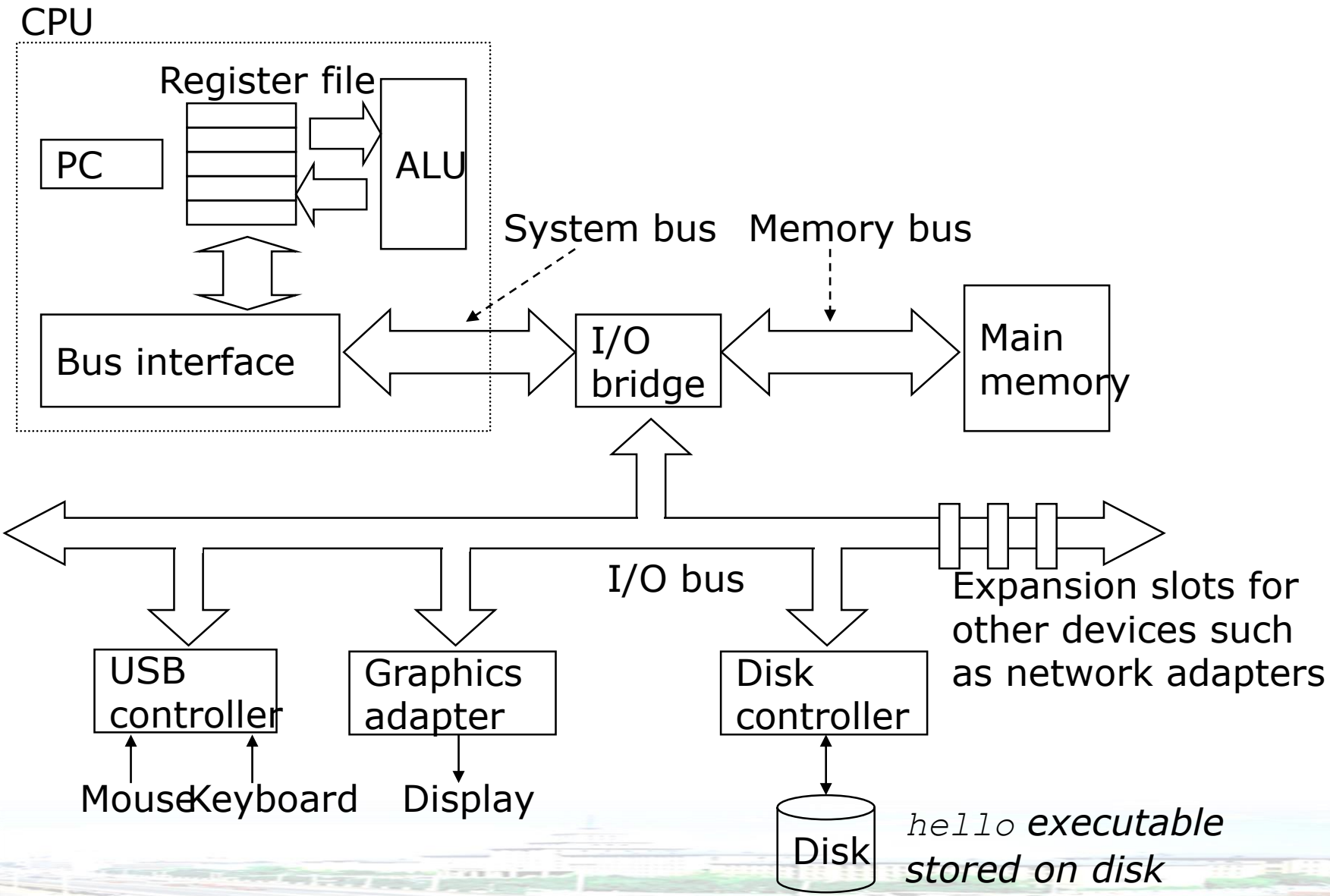
回顾下：
换入换出原则？



进程状态变化图（七状态）

挂起状态 - 引入主存←→外存的交换机制，虚拟存储管理的基础

Computer Architecture



第7章 虚拟内存

主要内容

7.1 背景

（1）内存不够用怎么办？ （2）内存管理视图

（3）用户眼中的内存 （4）虚拟内存的优点

7.2 虚拟内存实现--按需调页（请求调页）

（1）交换与调页（2）页表的改造（3）请求调页过程

7.3 页面置换

（1）FIFO页面置换

（2）OPT（最优）页面置换

（3）LRU页面置换（准确实现：计数器法、页码栈法）

（4）近似LRU页面置换（附加引用位法、时钟法）

7.4 其他相关问题

（1）写时复制 （2）交换空间（交换区）与工作集

（3）页置换策略：全局置换和局部置换

（4）系统颠簸现象和Belady异常现象

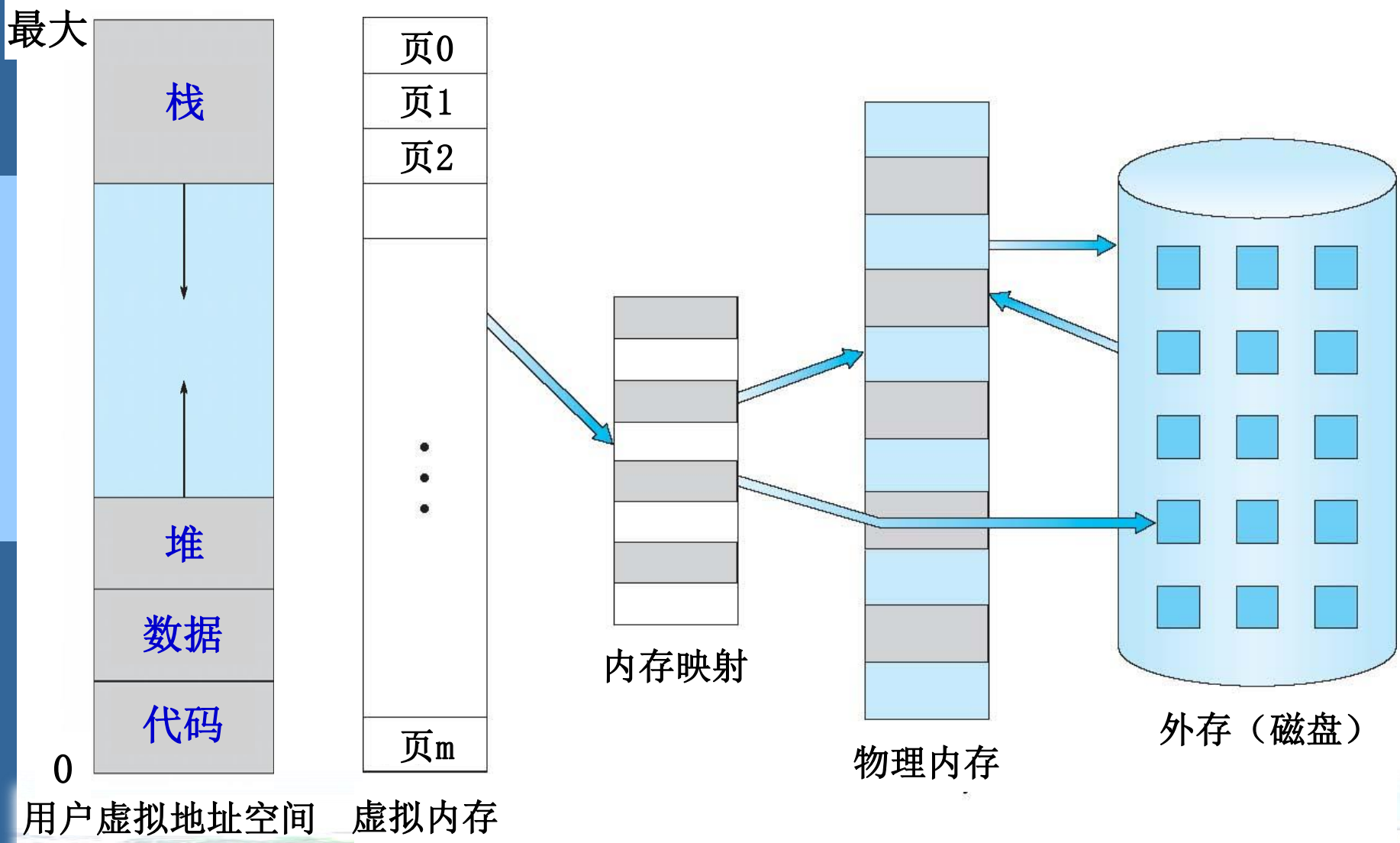
（5）虚拟内存中程序优化

7.5 CSAPP第9章<虚拟内存>串讲

7.1 背景

- 内存不够用怎么办?
- 内存管理视图
- 用户眼中的内存
- 虚拟内存的优点

内存不够怎么办？



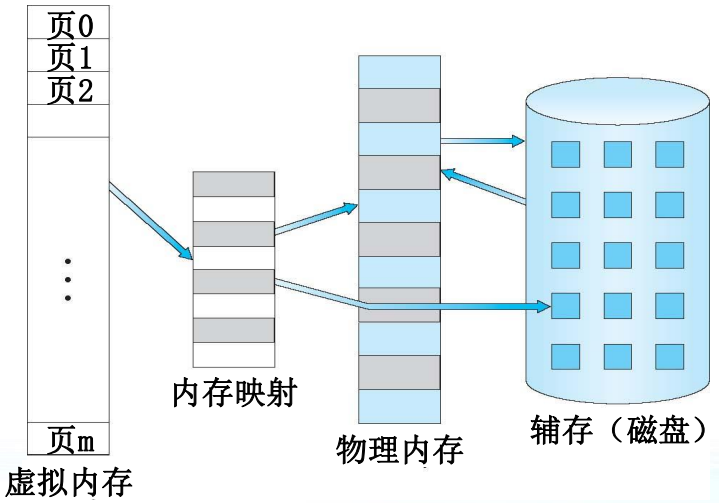
思考两个小问题

一个实际的问题：

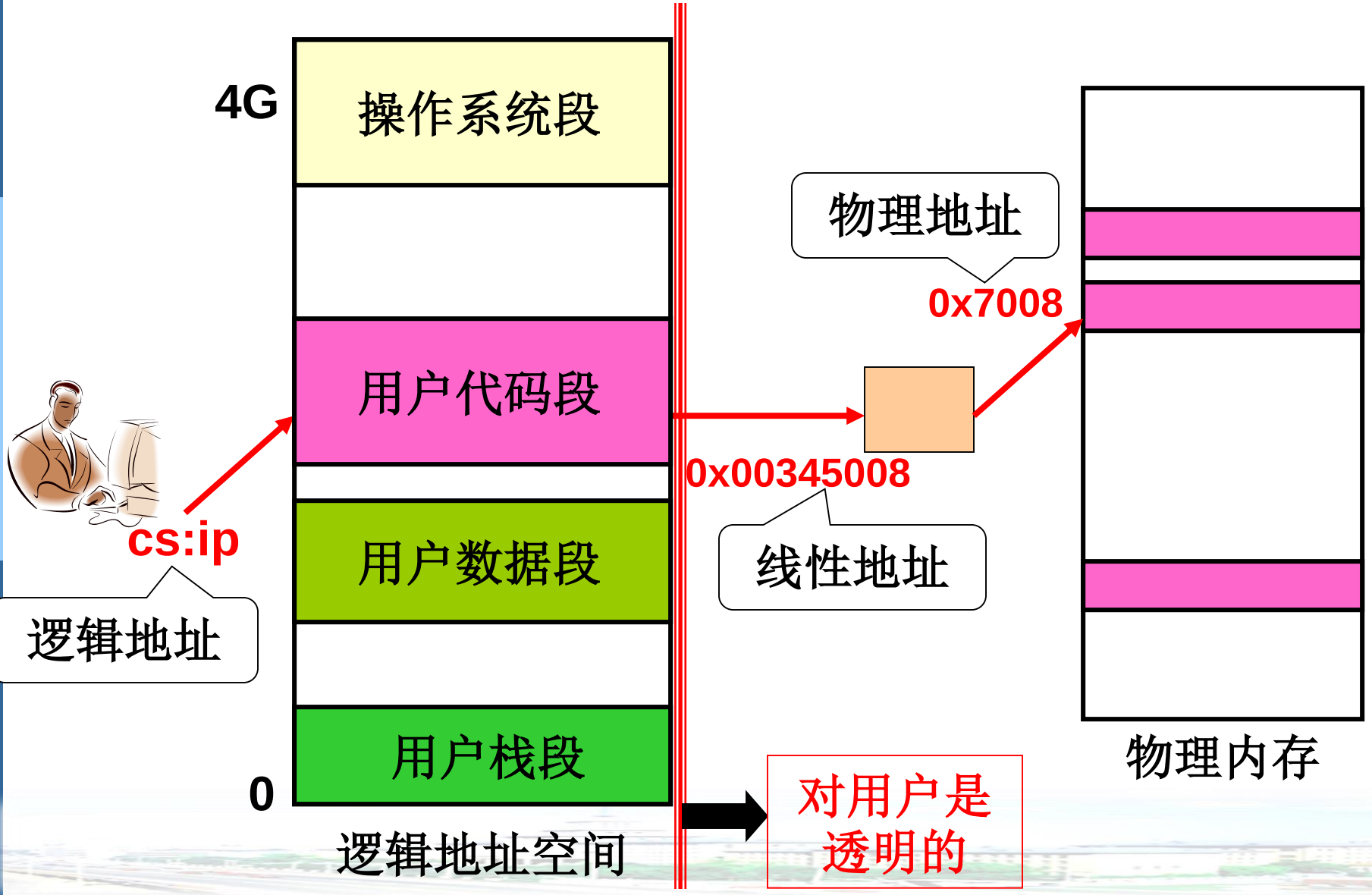
用word打开一个几百页的文档（至少几百兆）。在打开，前后拖拽；长时间不用或停留在某个页（操作），再拖拽。几种情况下，会发生什么现象，操作系统要做什么？

另一个实际的问题：

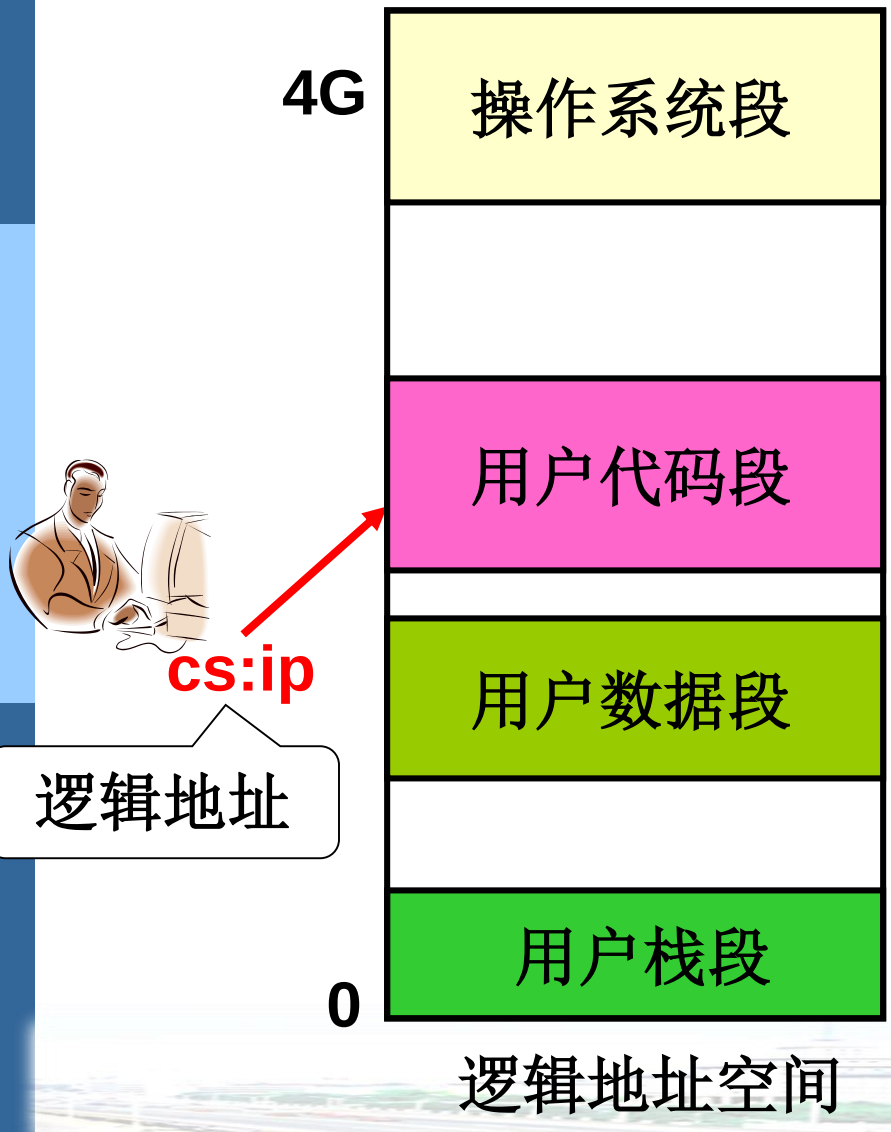
用word打开几个几百页的文档包含大量图片（至少几百兆）。在打开，前后拖拽，文档间切换时，会发生什么现象，操作系统要做什么？



内存管理视图


4G
操作系统段
用户代码段
用户数据段
用户栈段
0
逻辑地址空间
cs:ip
逻辑地址
0x00345008
线性地址
物理地址
0x7008
物理内存
对用户是
透明的

用户眼里的内存!

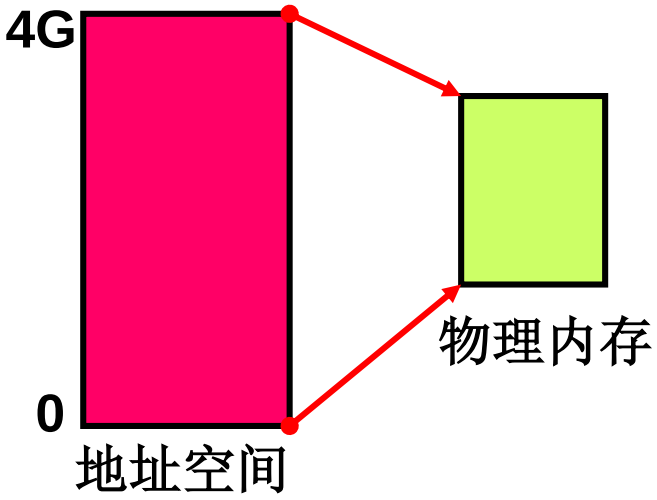


- 1个4GB(很大)的地址空间
- 用户可随意使用该地址空间，就象单独拥有4G内存
- 这个地址空间怎么映射到物理内存，用户全然不知

必须映射，否则不能用!

- 这种内存管理和使用方式被称为“虚拟内存”

虚拟内存的优点

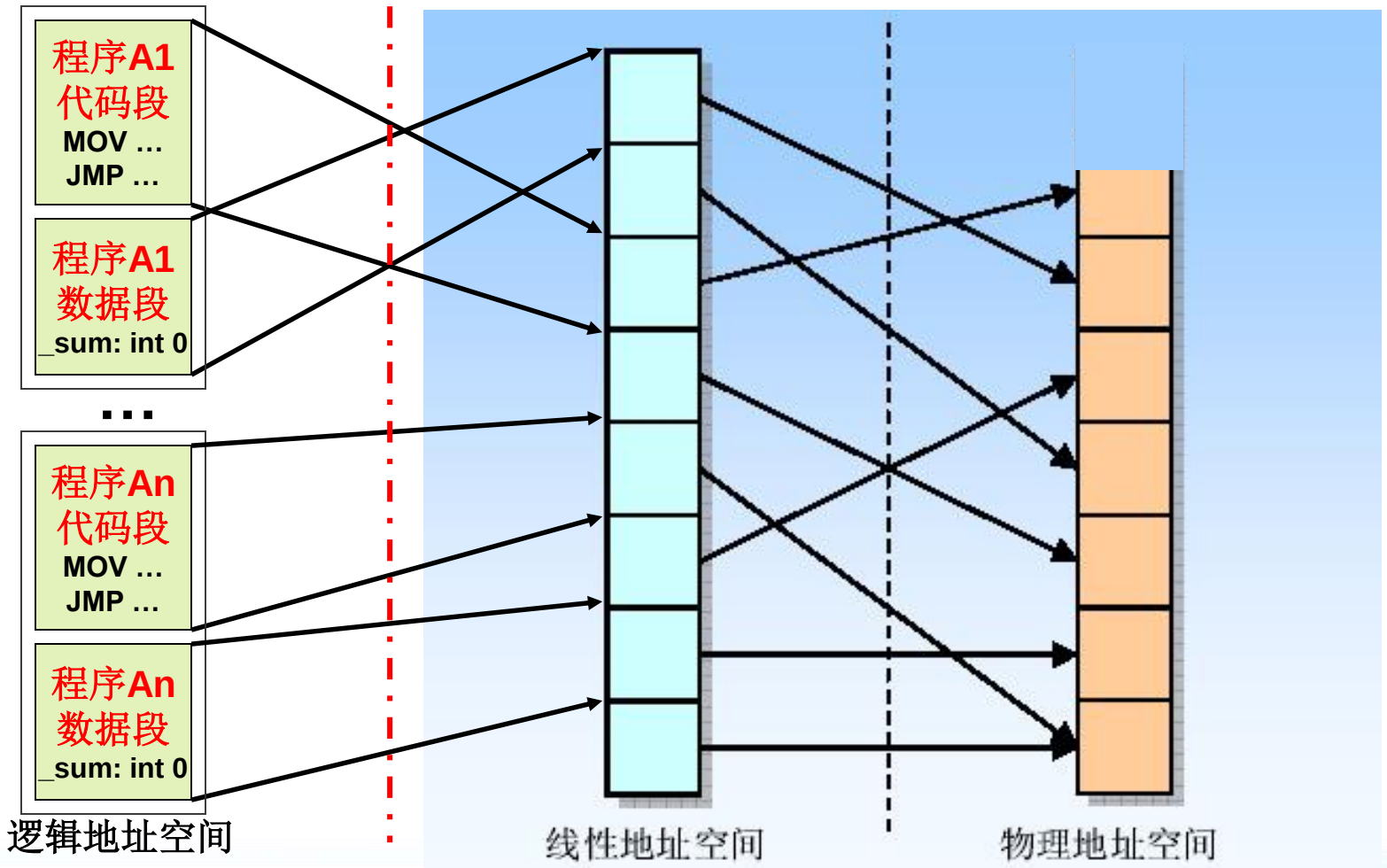


- 优点1: 地址空间>物理内存
  - 用户可以编写比内存大的程序
  - 4G空间可以使用，简化编程
- 优点2: 部分程序或程序的部分放入物理内存
  - 内存中可以放更多进程，并发度好，效率高
  - 将需要的部分放入内存，有些用不到的部分从来不放入内存，内存利用率高
  - 程序开始执行、响应时间等更快

如一些处理异常的代码!

虚拟内存思想既有利于系统，又有利于用户

回顾地址空间


程序A1
代码段
MOV …
JMP …
程序A1
数据段
_sum: int 0
…
程序An
代码段
MOV …
JMP …
程序An
数据段
_sum: int 0
逻辑地址空间
线性地址空间
物理地址空间

小结

什么是虚拟内存？

虚拟内存使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

基于局部性原理,在程序装入时,可以将程序中的很快会用到的部分装入内存,暂时用不到的部分留在外存,其包含如下四方面技术手段：

段页管理 部分加载 按需调页 换入换出

实际效果是：每个进程都能占有使用CPU可寻址（32位机4G）的线性地址空间，其可以远大于物理内存空间。

7.2 按需调页（请求调页）

如何实现虚拟内存？

（1）交换与调页

（2）页表的改造

（3）请求调页过程

从交换到调页

早期

（1）整个程序装入内存执行，内存不够不能运行

（2）内存不足时以进程为单位在内外存之间交换

（3）调页，也称惰性交换，以页为单位在内外存之间交换

（4）请求调页，也称按需调页，即对不在内存中的“页”，当进程执行时要用时才调入，否则有可能到程序结束时也不会调入

现在

从段页式内存管理开始

段号+偏移(cs:ip)

逻辑地址

段号	基址	长度	保护
0	0x4000	0x0800	R
1	0x4800	0x1400	R/W
2	0xF000	0x1000	R/W
3	0x0000	0x3000	R

部分逻辑地址对应段表项，发现缺段后调入

线性地址

页号	偏移

部分线性地址对应页表项，发现缺页后调入

页框号	保护
5	R
1	R/W
3	R/W
7	R

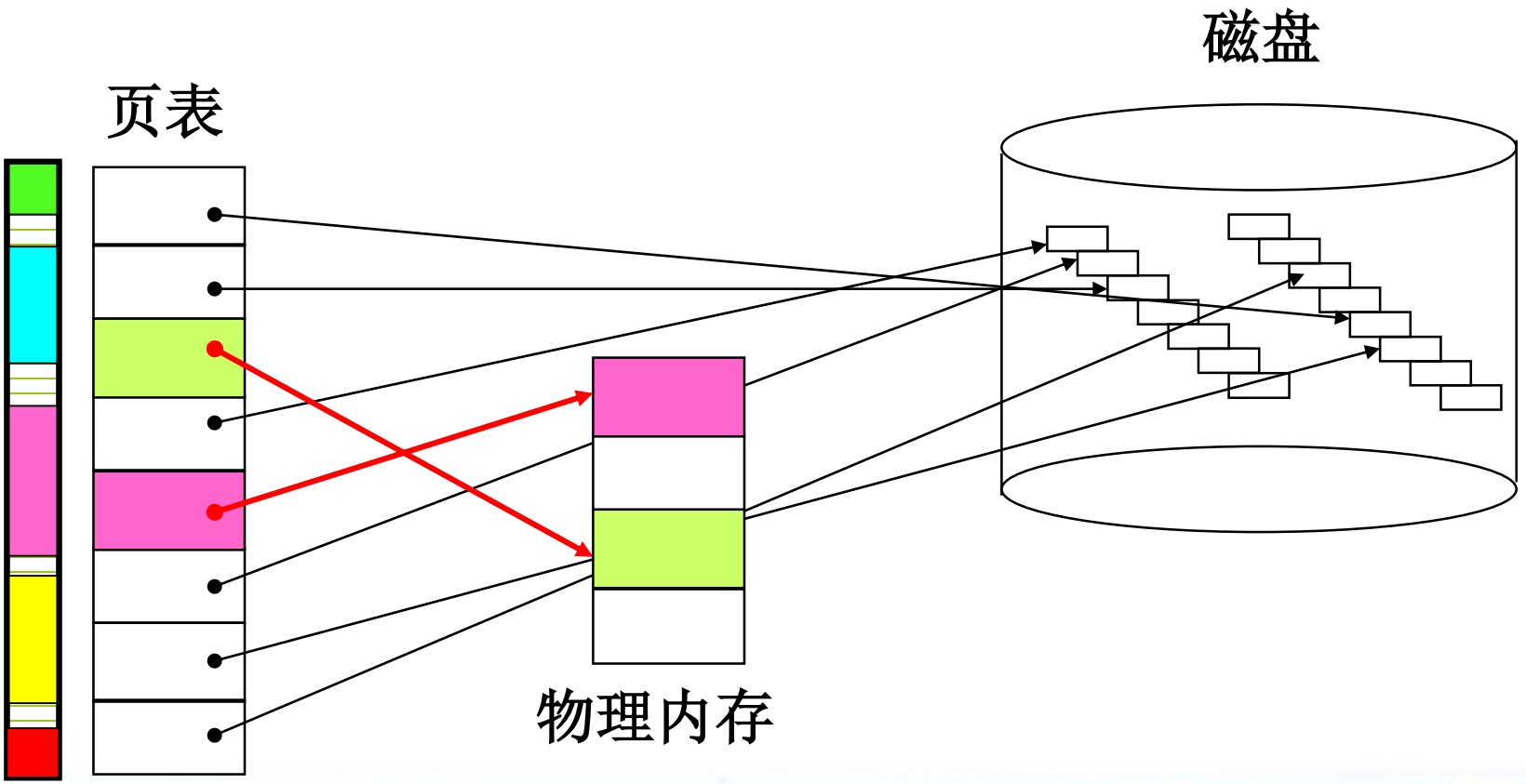
分页易于硬件实现、对用户透明，适合请求调入

物理页号	偏移

物理地址

虚拟内存中的页面映射关系

- 部分线性地址(逻辑页)对应物理页，其它（段中）页呢?



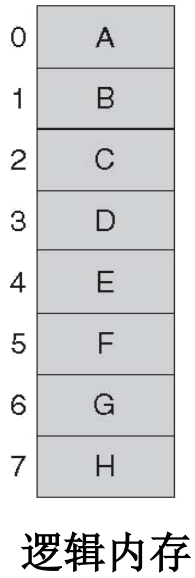
如何记录页是否在内存？

帧号　有效/无效位

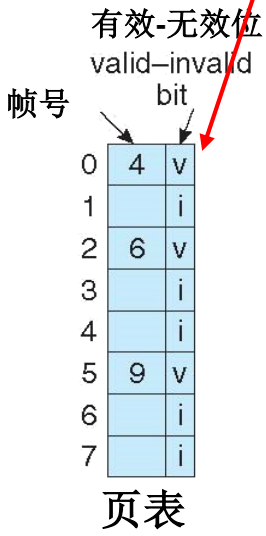
	v
	v
	v
	v
	i
….	
	i
	i

改造后的页表

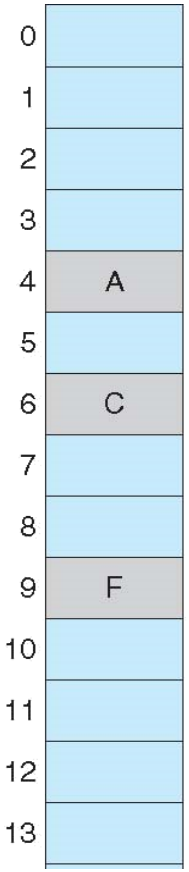
某些页不在内存中时的页表



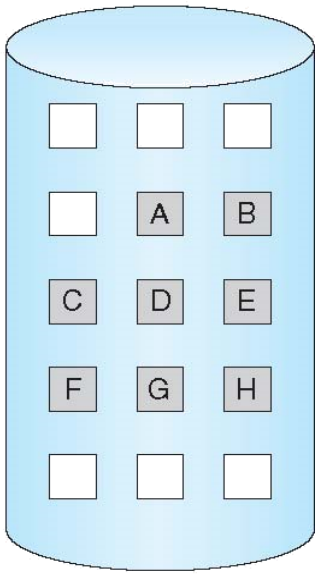
逻辑内存



页表



物理内存

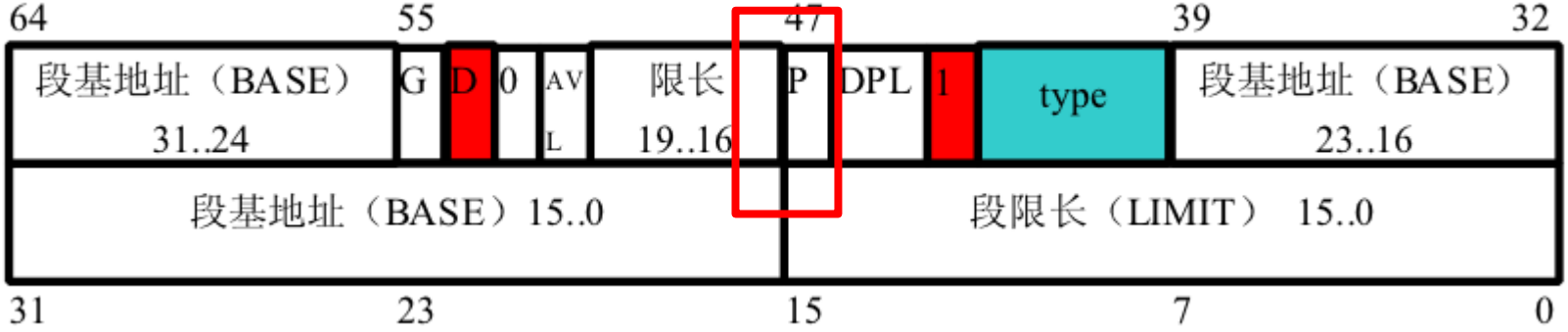


辅存（磁盘）

⑩ 改造页表，页表项增加“有效/无效位”

回顾：Intel x86的分段硬件 – LDT、GDT

- 段描述符: LDT(GDT)中的表项



	TYPE			说明
十进制值		E	W	A	数据段
0	0	0	0	0	只读
1	0	0	0	1	只读、已访问
2	0	0	1	0	读写
3	0	0	1	1	读写、已访问
4	0	1	0	0	只读、向下扩展
5	0	1	0	1	只读、向下扩展、已访问
6	0	1	1	0	读写、向下扩展
7	0	1	1	1	读写、向下扩展、已访问

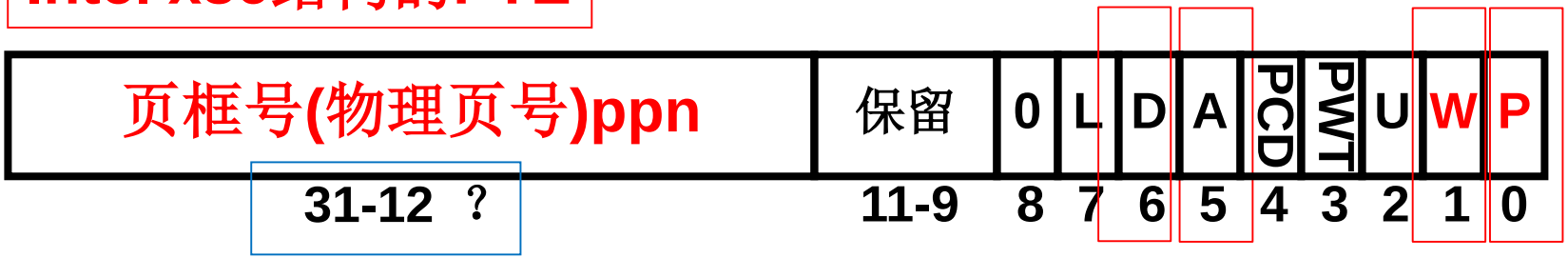
32位保护模式下的寻址方式

	TYPE				说明
十进制值		C	R	A	代码段
8	1	0	0	0	只执行
9	1	0	0	1	只执行、已访问
10	1	0	1	0	执行、可读
11	1	0	1	1	执行、可读、已访问
12	1	1	0	0	只执行、一致
13	1	1	0	1	只执行、一致、已访问
14	1	1	1	0	执行、可读、一致
15	1	1	1	1	执行、可读、一致、已访问

P表示是否在内存，A表示是否已访问

回顾：Intel x86的分页硬件

Intel x86结构的PTE



P--位0是存在（Present）标志

A--位5是已访问（Accessed）标志。

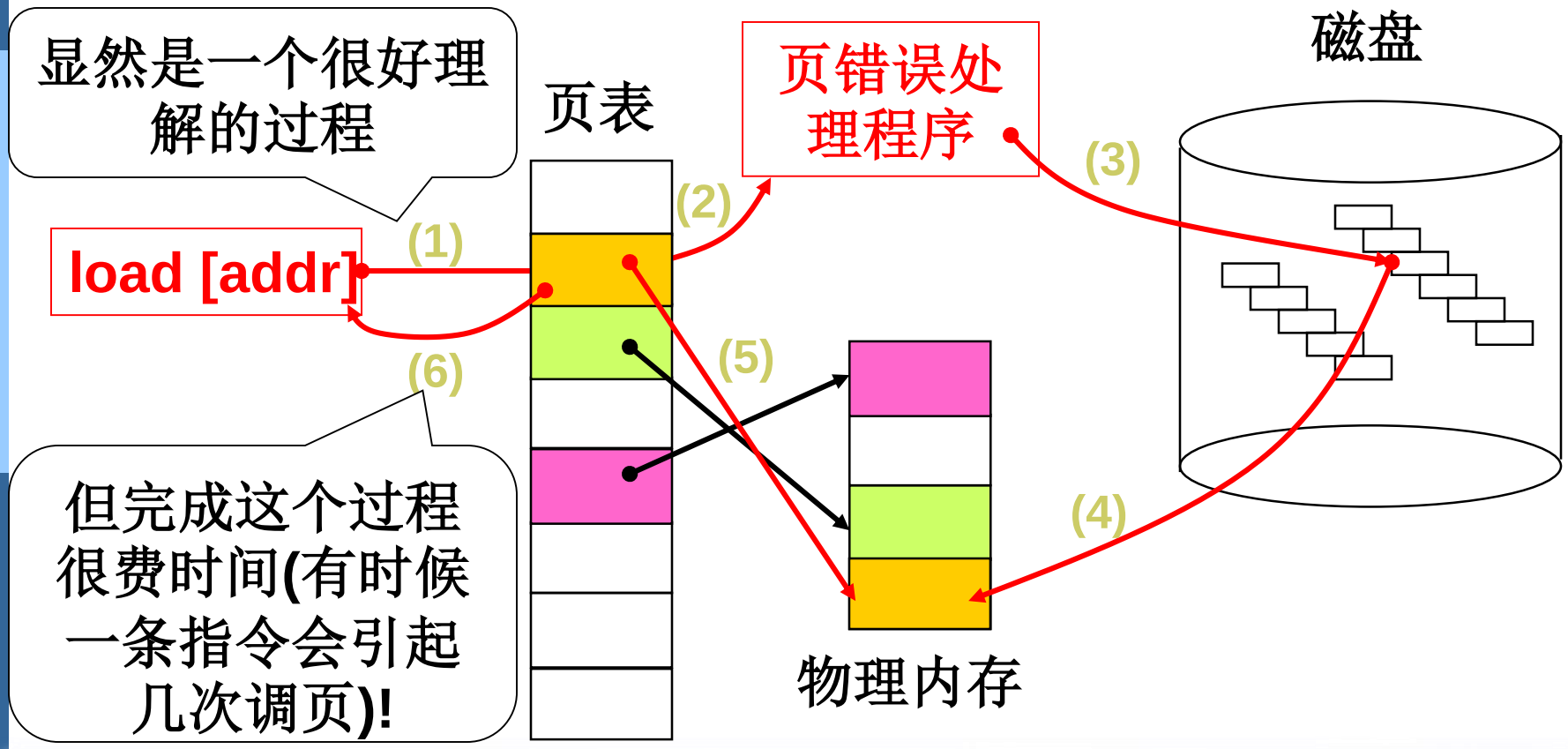
D--位6是页面已被修改（Dirty）标志。

R/W--位1是读/写（Read/Write）标志。

换出到SWAP分区后地址存到哪里？可以存到页框号的位置

请求调页过程

❿ 当访问没有映射的线性地址时…

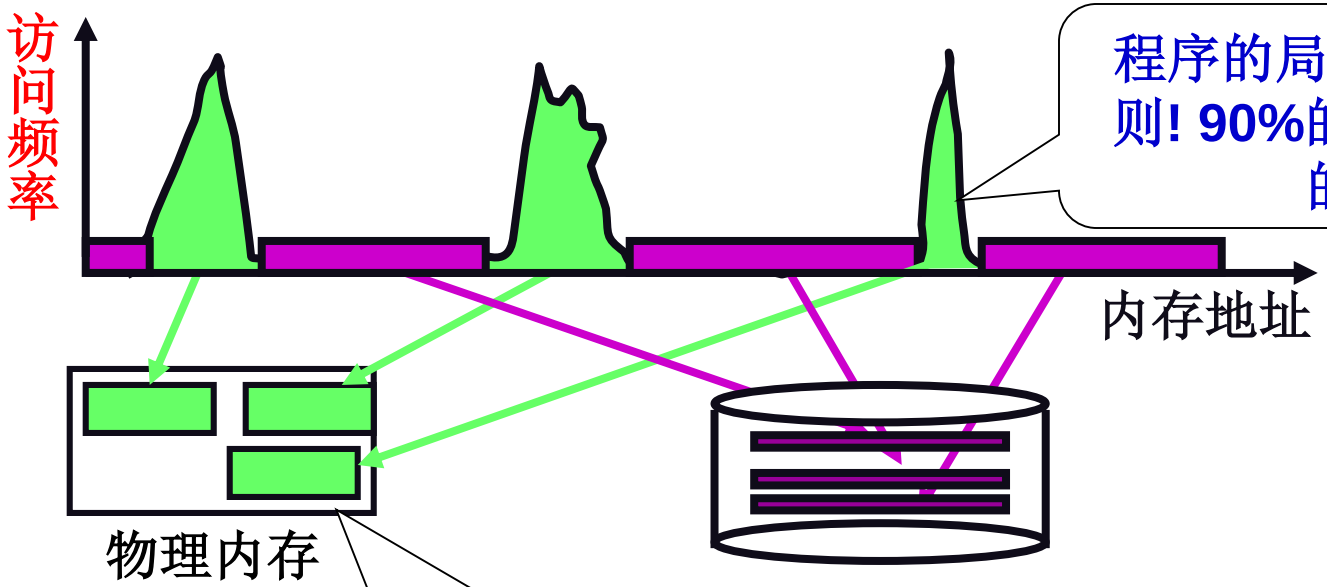


请求调页为什么可行?

⑩ 分配给一个进程的物理页框数应该足够多!

这又违背了分页和请求调页的原则，又需要折衷!

- 如何设计这些参数? 再从计算机的基本特征开始!



程序的局部性：90/10原则! 90%的程序访问10%的地址!

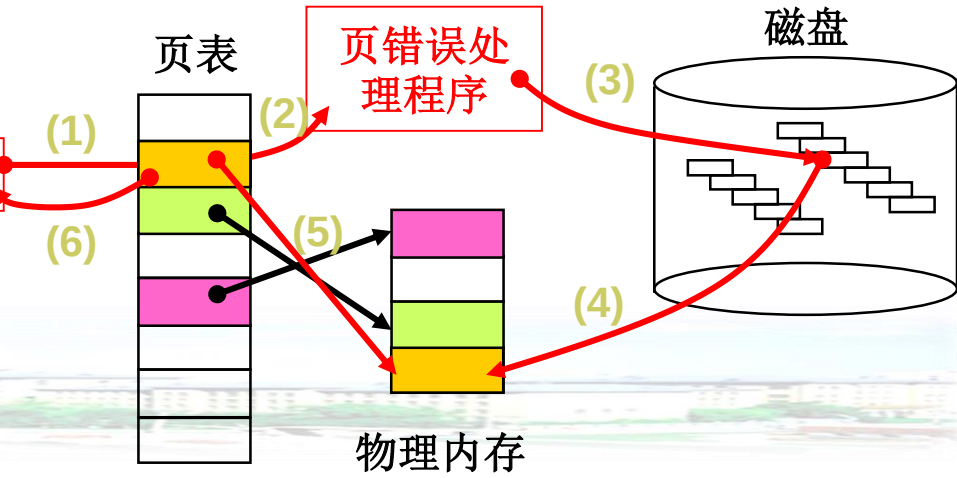
页面4K，调入一页后许多指令不出现页错误

- 这些参数从实践中获得!

请求调页的具体实现细节

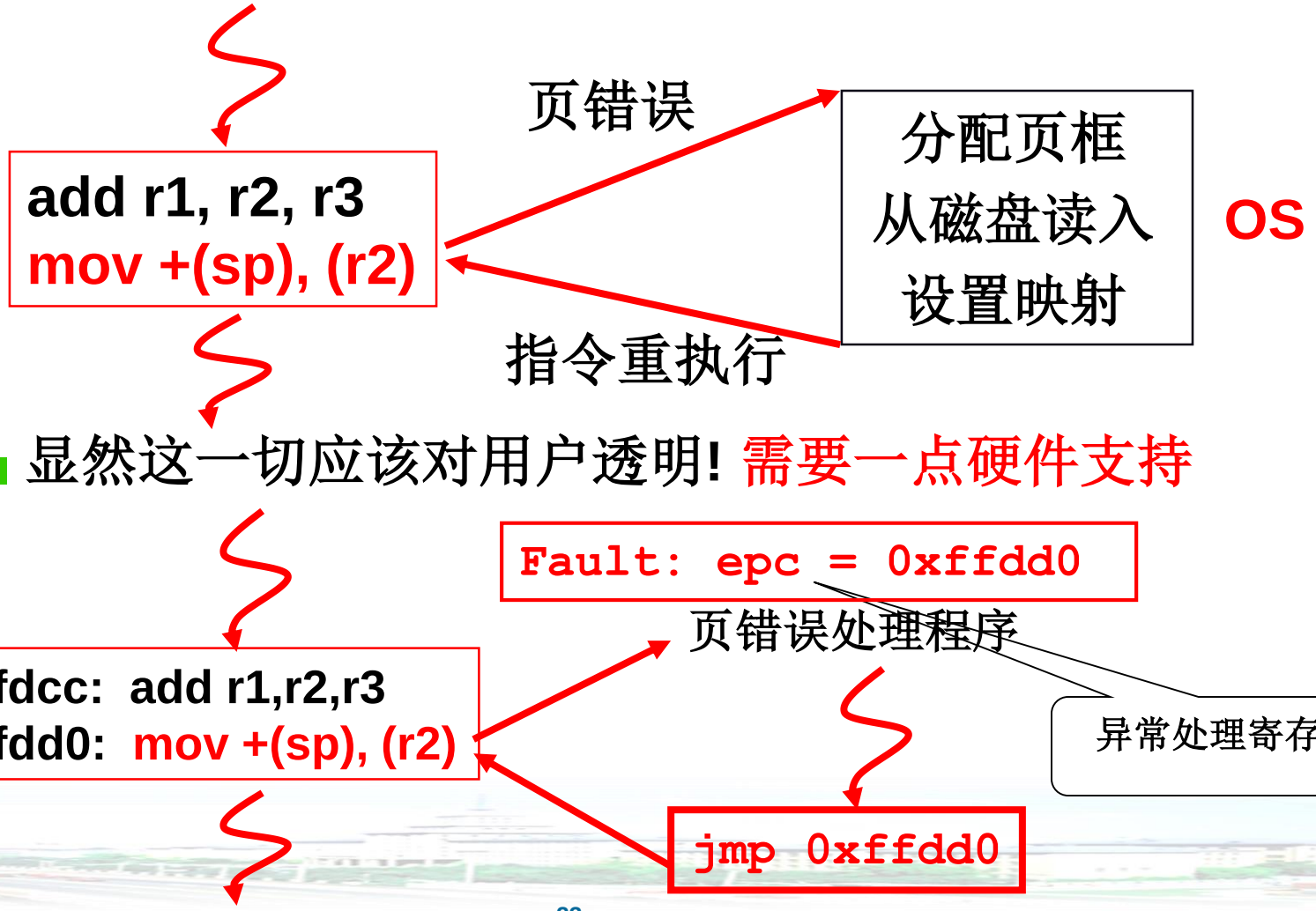
- (1): load [addr]，而addr没有映射到物理内存
  - 根据addr查页表(MMU)，页表项的P位为0，引起缺页中断(page fault)
- (2): 设置“缺页中断”程序
- (3): “缺页中断处理程序”需要读磁盘
- (4): 选一个空闲页框
- (5): 修改页表
- (6): 重新开始指令

学过磁盘处理后自然就明白了!



如何重新开始指令?

⑩ 在指令执行过程中出现页错误



如何选一个空闲页框？ 页面淘汰(置换)

⑩ 没有空闲页框怎么办？分配的页框数是有限的

- 需要选择一页淘汰
- 有多种淘汰选择。如果某页刚淘汰出去马上又要用…

- **FIFO**，最容易想到，怎么评价?
- 有没有最优的淘汰方法，**OPT(MIN)**（**OPT-Optimal**）
- 最优淘汰方法能不能实现，能否借鉴思想，**LRU**

- 再来学习几种经典方法，它可以用在许多需要淘汰(置换)的场合…

7.3 页面置换

（1）FIFO页面置换

（2）OPT（最优）页面置换

（3）LRU页面置换

准确实现：计数器法、页码栈法

（4）近似LRU页面置换

附加引用位法、时钟法

FIFO页面置换

⑩ FIFO算法:先入先出，即淘汰最早调入的页面

- 一实例: 分配了3个页框(frame)，页面引用序列为

A B C A B D A D B C B

D换A不太合适!

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- 评价准则: 缺页次数；本实例，FIFO导致7次缺页
- 选A、B、C中未来最远将使用的，是理想办法！

OPT(MIN)页面置换

⑩ **OPT(MIN)**算法: 选未来最远将使用的页淘汰。是一种最优的方案，可以证明缺页数最小!

- 继续上面的实例: (3frame)A B C A B D A D B C B

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

- 本实例，MIN导致5次缺页
- 可惜，MIN需要知道将来发生的事… 怎么办?

LRU页面置换(OPT的可实现[illegible]算法)

Least-Recently-Used

- 用过去的历史预测将来。LRU算法: 选最近最长时间没有使用的页淘汰(也称最近最少使用)。
  - 继续上面的实例: (3frame)A B C A B D A D B C B

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  - 本实例，LRU也导致5次缺页

本例和OPT完全一样!

  - LRU是公认的很好的页置换算法，怎么实现?

LRU的准确实现方法1:计数器法

⑩ 每页维护一个时间戳(time stamp)，即计数器

具体方法：设1个时钟(计数)寄存器，每次页引用时，【所有页】计数器加1，并将该值复制到相应页表项中。当需要置换页时，选择计数值最小的页。

- 继续上面的实例: (3frame)A B C A B D A D B C B

time stamp

	A	B	C	A	B	D	A	D	B	C	B
A	1	1	1	4	4	4	7	7	7	7	7
B	0	2	2	2	5	5	5	5	9	9	11
C	0	0	3	3	3	3	3	3	3	10	10
D	0	0	0	0	0	6	6	8	8	8	8

选具有最小时间戳的页!

选A淘汰!

- 每次地址访问都需要修改时间戳，需维护一个全局时钟(该时钟溢出怎么办?)，需要找到最小值 … 这样的实现代价较大 ⇒ 几乎没人用

LRU准确实现方法2:页码栈法

⑩ 维护一个页码栈

具体方法：建立一个容量为有效帧数的页码栈。每当引用一个页时，该页号就从栈中上升到栈的顶部，栈底为LRU页。当需要置换页时，直接置换栈底页即可。

- 继续上面的实例: (3frame)A B C A B D A D B C B

	A	B	C	A	B	D	A	D	B	C	B
页码栈			C	A	B	D	A	D	B	C	B
		B	B	C	A	B	D	A	D	B	C
	A	A	A	B	C	A	B	B	A	D	D

选栈底页淘汰!

- 每次地址访问都需要修改栈，实现代价仍然较大 ⇒ LRU准确实现用的少

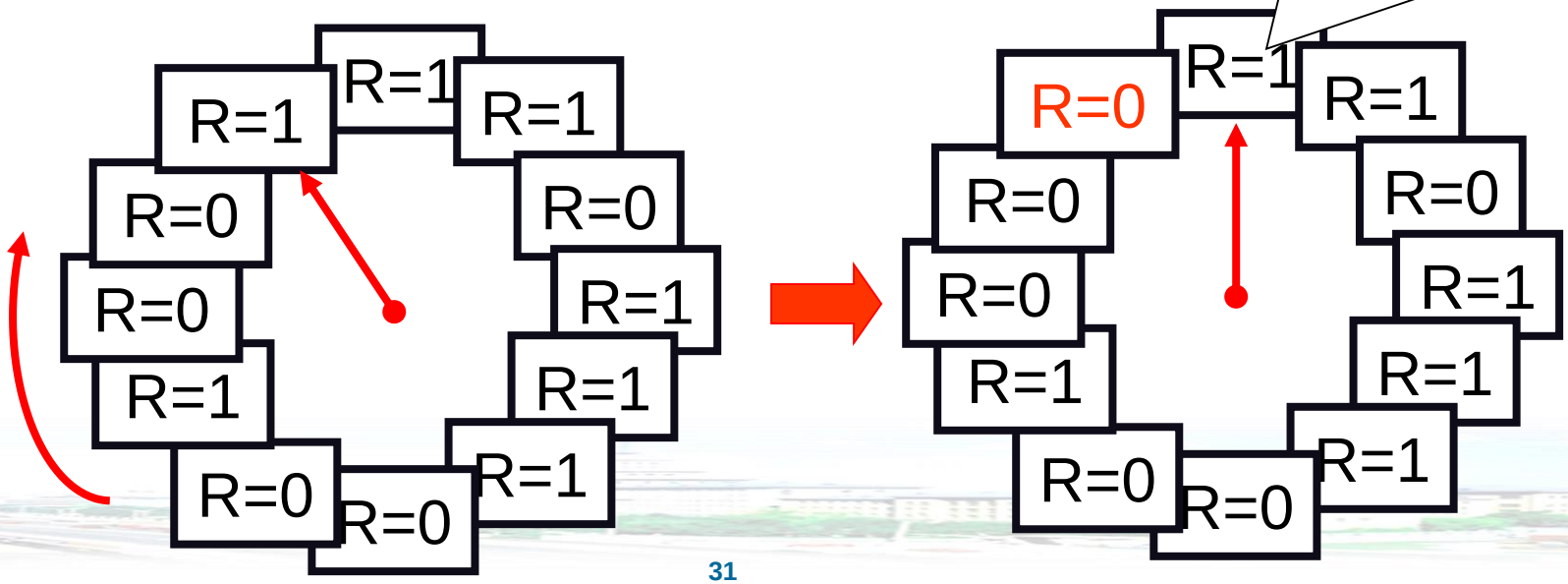
LRU近似实现 – 将时间计数变为是和否

- 每个页加一个引用位(reference bit)
  - 每次访问一页时，硬件自动设置该位为1
  - 选择淘汰页：扫描该位，是1时清0，并继续扫描；直到碰到是0时淘汰该页，记录该位置，下次继续

再给一次机会(Second Chance Replacement)

组织成循环队列较合适！

SCR实现方法称为Clock Algorithm



Clock算法实例

- 继续上面的实例: (3frame)A B C A B D A D B C B

（1）将可用帧按顺序组成环形队列，引用位初始置0，并置指针初始位置；

（2）缺页时从指针位置扫描引用位，将1变0，直到找到引用位为0的页；

（3）当为某页初始分配页框或页被替换，则指针移到该页的下1页。

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A*	A*	A*	A*	A*	D*	D*	D*	D*	C*	C*
2		B*	B*	B*	B*	B	A*	A*	A*	A	A
3			C*	C*	C*	C	C	C	B*	B	B*

引用位！

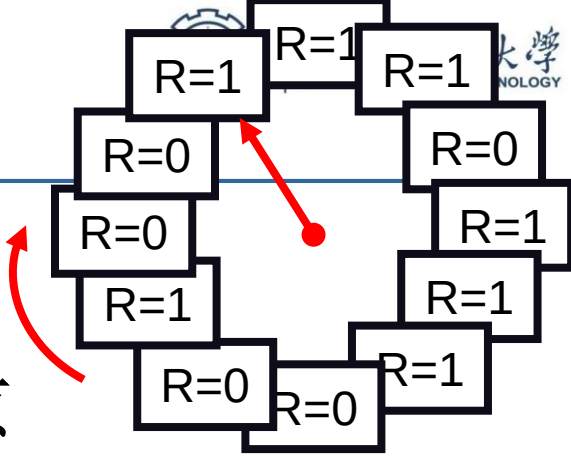
扫描指针！

扫了一圈，A,B,C 引用位均清除

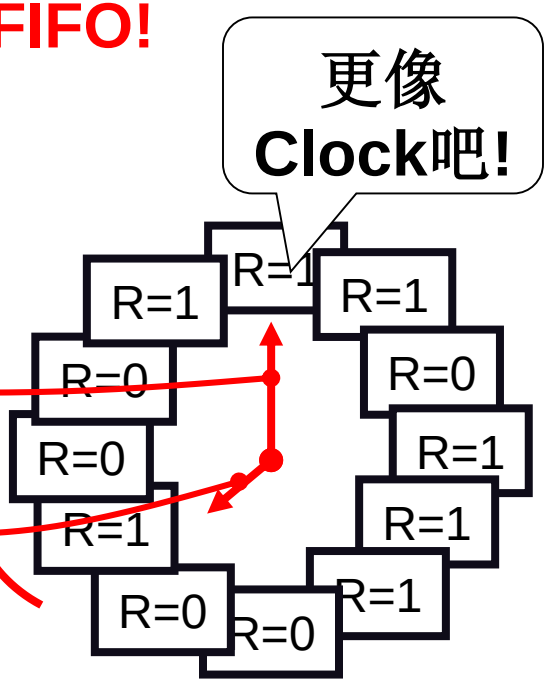
- 本实例，Clock算法也导致7次缺页
- Clock算法是公认的很好的近似LRU的算法

随堂习题以及对比分析

Clock算法分析的改造



- 如果缺页很少，会？ 所有的R=1
  - hand scan一圈后淘汰当前页，将调入页插入hand位置，hand前移一位 退化为FIFO!
  - 原因: 记录了太长的历史信息… 怎么办?
  - 定时清除R位… 再来一个扫描指针!



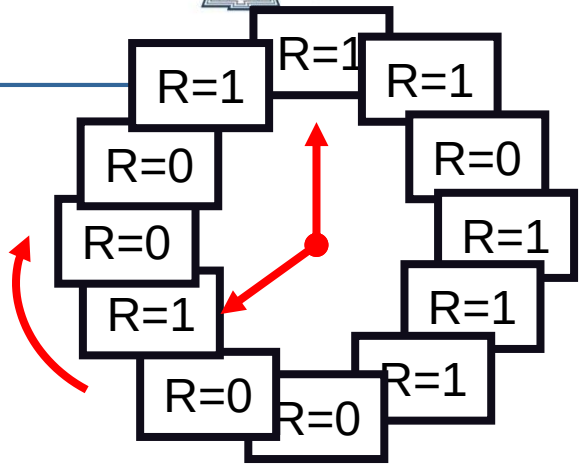
指针2：用来清除R位，移动速度要快!

指针1：用来选择淘汰页，移动速度慢!

指针1用来选择淘汰页，缺页时用；
指针2根据设定的时间间隔定时清除R位

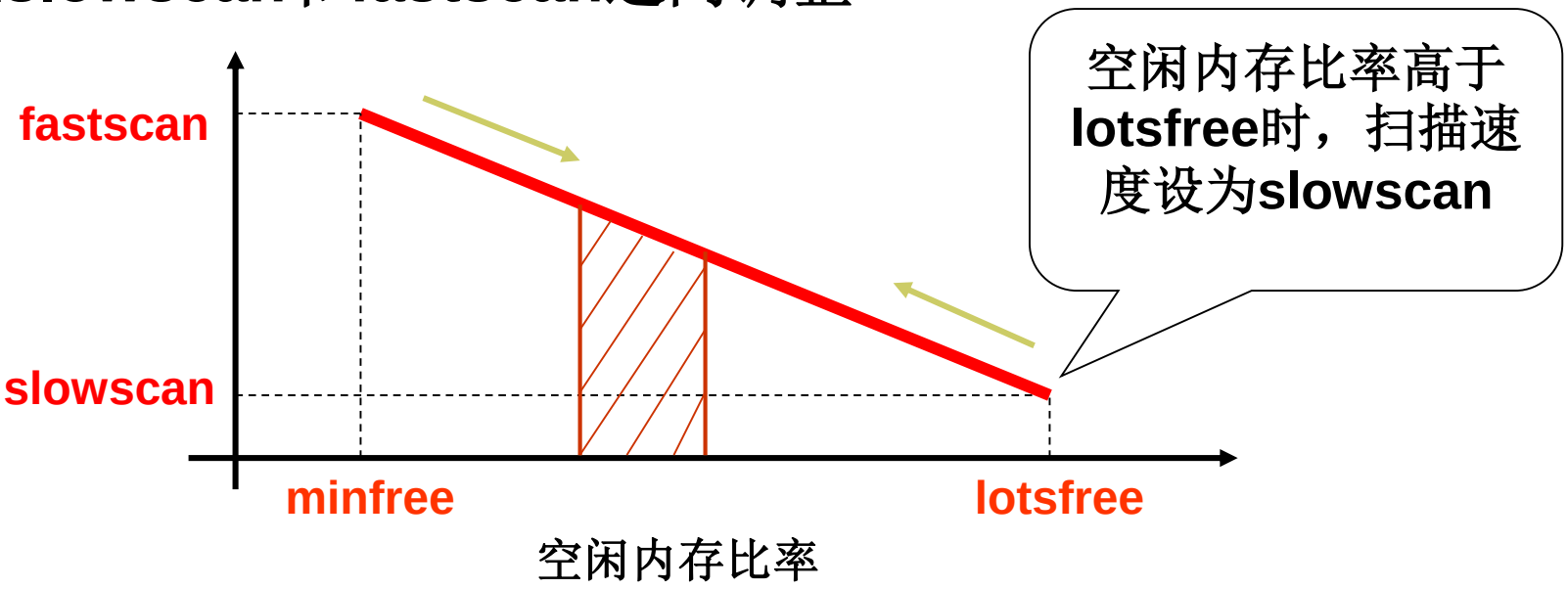
- 清除R位的hand如何定速度，若太快? 又成了FIFO!

Solaris实际做法



⑩ 设定值吗？ 系统负载并不固定…

- 在slowscan和fastscan之间调整



Clock置换算法总结

(1)简单的clock置换算法:

- 每页设置一位访问位。当某页被访问了，则访问位→置“1”；
- 内存中的所有页链接成一个循环队列；

-Clock置换算法流程：

- A:如当前查询指针所在页命中，访问位→置“1”，查询指针保持不动；
- B:否则，循环检查各页面的使用情况。
  - 1、若访问位为“0”，选择该页淘汰，查询指针前进一步；
  - 2、若访问位为“1”，复位访问位为“0”，查询指针进一步。

- 又称为“最近未使用”置换算法（NRU）

改进型Clock置换算法

(2)改进型Clock置换算法:

访问位A、修改位M，共同表示一个页面的状态

页面的四种状态:

- 第一类 00：(A=0;M=0)最近未被访问也未被修改
- 第二类 01：(A=0;M=1)最近未被访问但已被修改
- 第三类 10：(A=1;M=0)最近已被访问但未被修改
- 第四类 11：(A=1;M=1)最近已被访问且被修改

Intel x86结构的PTE

改进型Clock置换算法

(2)改进型Clock置换算法:

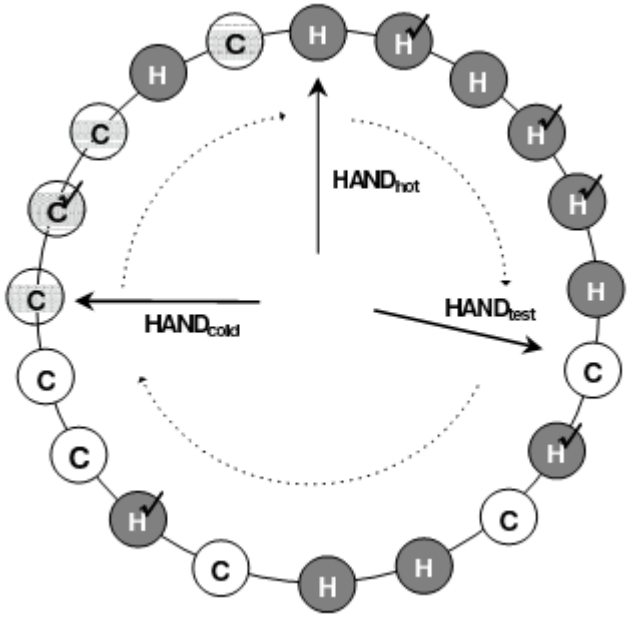
三轮扫描“循环队列”:

- **第一轮:** 查找A=0,M=0页面，若找到，替换；否则，进入下一步；
- **第二轮:** 查找A=0,M=1页面，若找到，替换；并把遍历过的页面的A位复位为“0”；若一直没找到，进入下一轮；
- **第三轮:** 把所有页面的A位复位为“0”，重复第一轮，必要时再重复第二轮。

拓展阅读

CAR: Clock with Adaptive Replacement @FAST 2004

CLOCK-Pro: An Effective Improvement of the CLOCK Replacement @USENIX ATC 2005



拓展阅读(续)

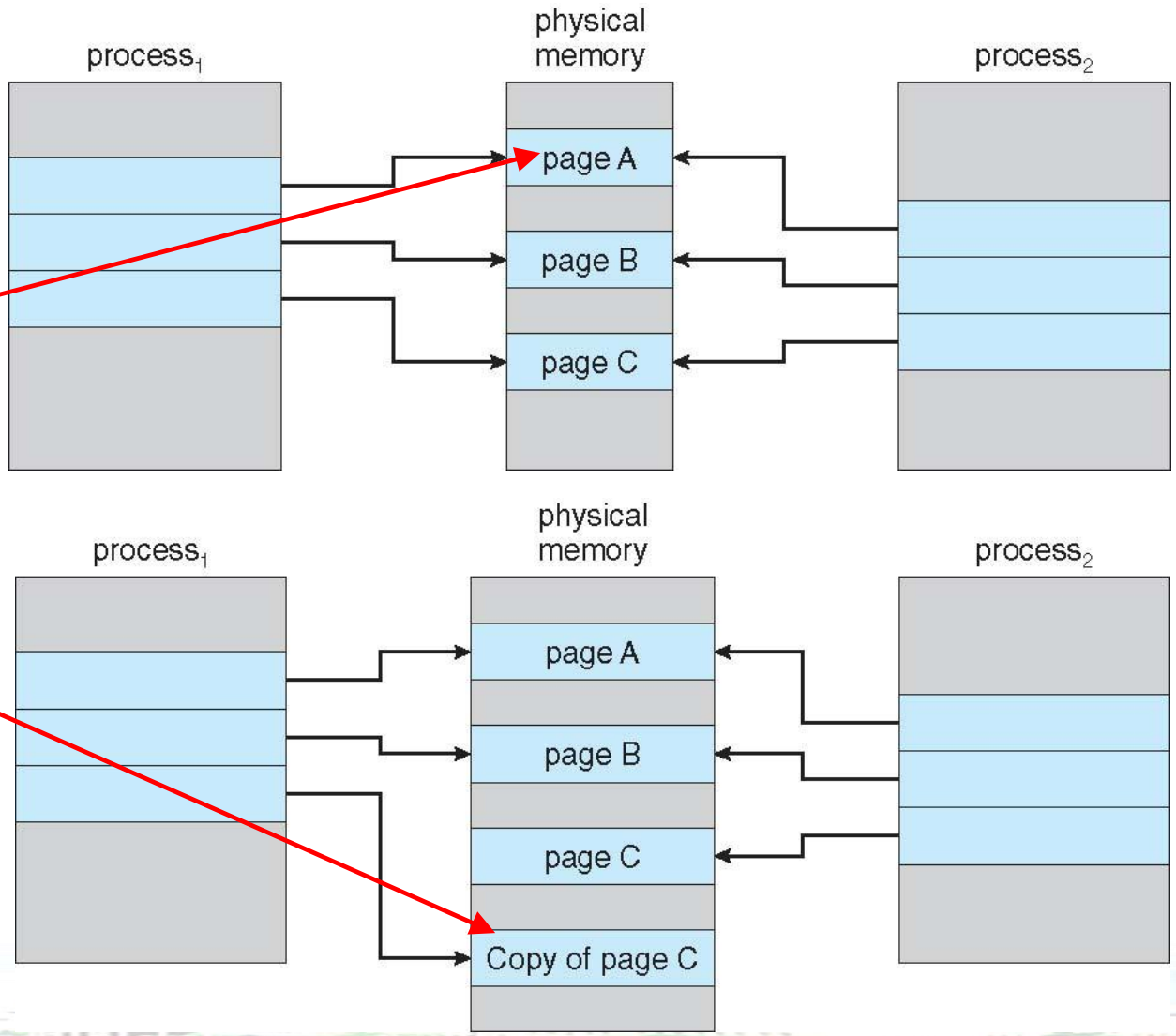
- Efficient SSD Caching by Avoiding Unnecessary Writes using Machine Learning @ICPP 2018
- LIPA: A Learning-based Indexing and Prefetching Approach for Data Deduplication@MSST 2019
- A Survey of Machine Learning Applied to Computer Architecture Design @ https://arxiv.org/

7.4 其他相关问题

(1) 写时复制
(2) 交换空间（交换区）与工作集
(3) 页置换策略：全局置换和局部置换
(4) 系统颠簸现象和Belady异常现象
(5) 虚拟内存中程序优化

写时复制

为了更快创建进程，子进程共享父进程的地址空间，仅当某进程要写某些页时，才为其复制产生一个新页



交换空间--页面置换到什么地方

换出的页面存到什么地方？

磁盘交换空间

- 基于普通文件系统：windows中pagefile.sys文件
- 独立的磁盘分区—生磁盘（RAW），不需要文件系统和目录结构，如linux中的swap分区

工作集

工作集（驻留集）：给进程分配的主存物理空间。是动态变化的。

⑩操作系统决定给特定的进程分配多大的主存空间？这需要考虑以下几点：

a) 分配给一个进程的存储量越小，在任何时候驻留在主存中的进程数就越多，从而可以提高处理器的利用率。
b) 如果一个进程在主存中的帧数过少，尽管有局部性原理，页错误率仍然会相对较高。
c) 如驻留集过大，由于局部性原理，给特定的进程分配更多的主存空间对该进程的错误率没有明显的影响。

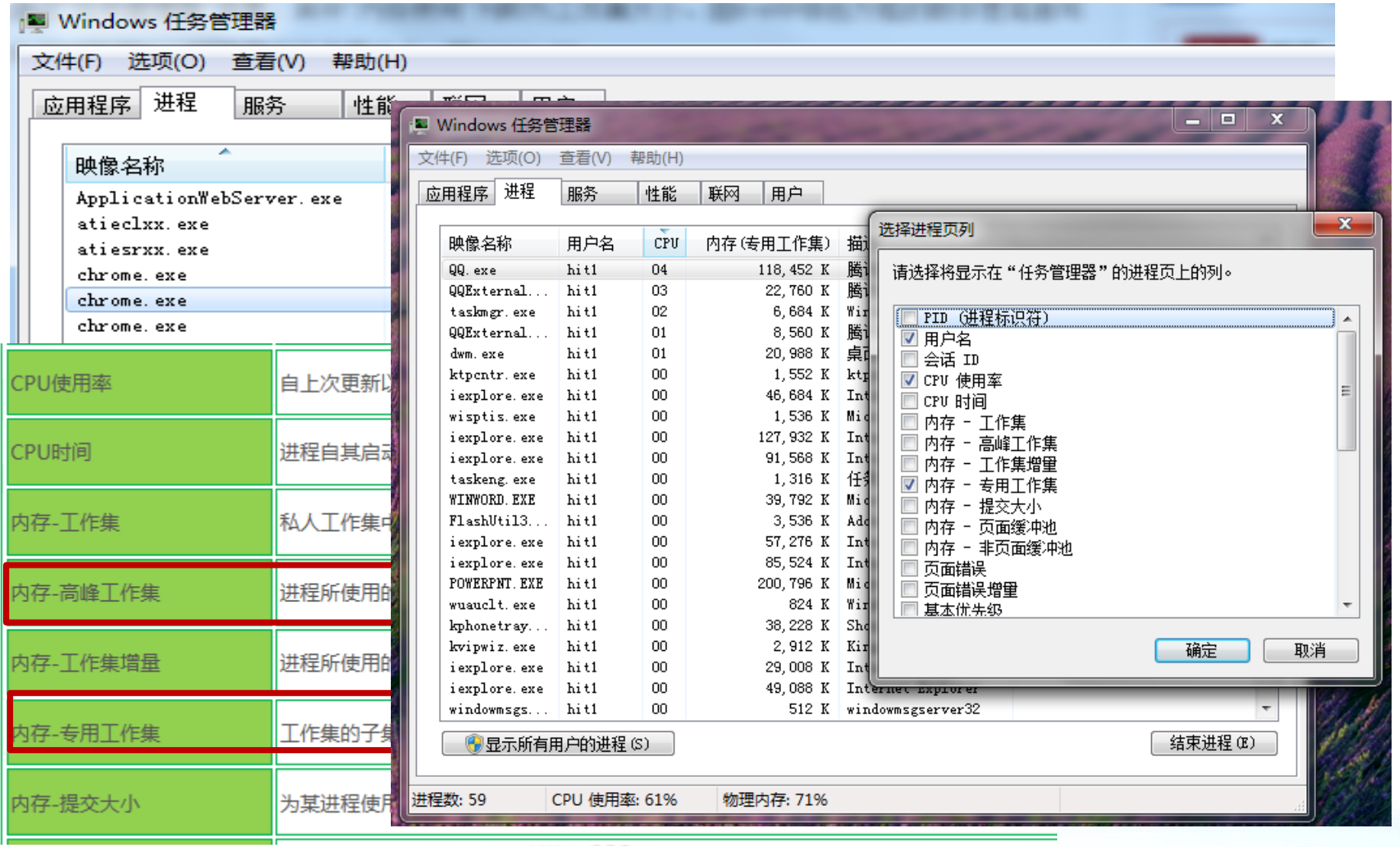
工作集分配

分为两种方式：

固定分配(fixed-allocation)：工作集大小固定，可以：各进程平均分配，根据程序大小按比例分配，按优先权分配。

可变分配(variable-allocation)：工作集大小可变，按照缺页率动态调整（高或低—>增大或减小常驻集），性能较好。增加算法运行的开销。

Windows工作集分配



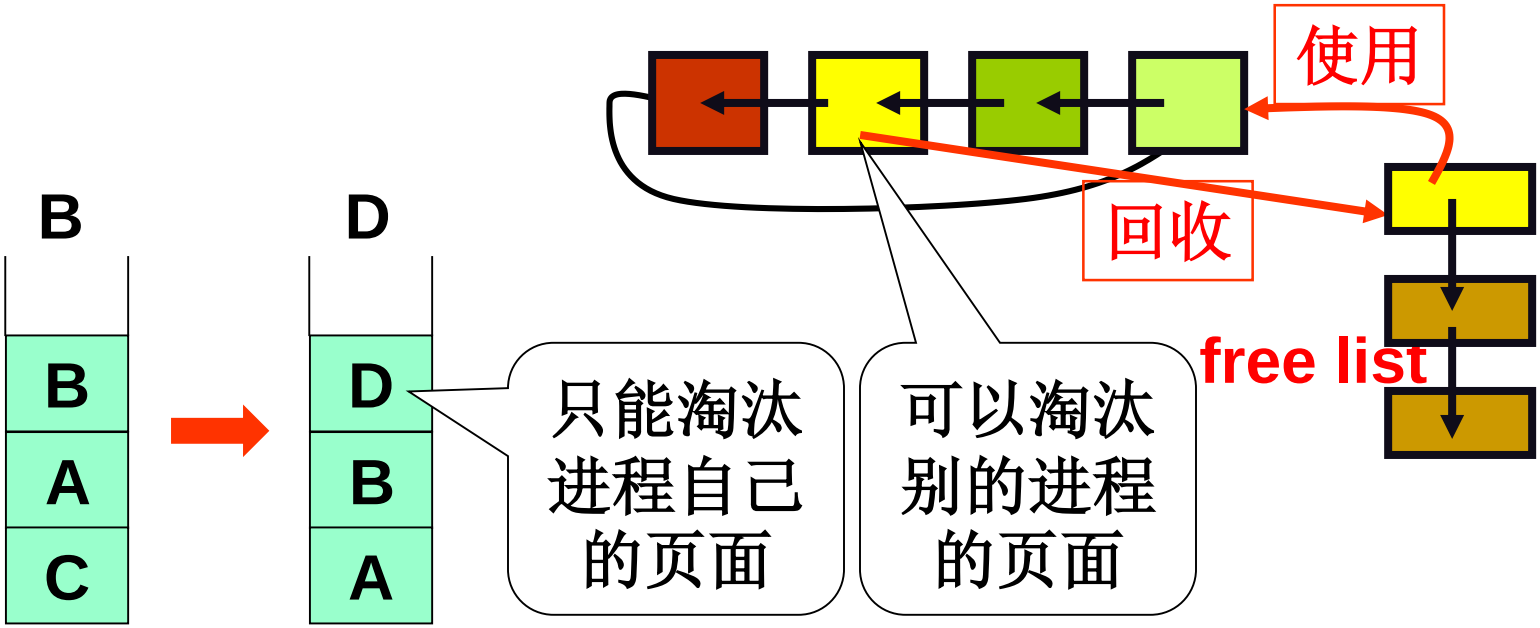
linux工作集

Ps命令是进程查看命令

RSS进程使用的驻留集大小或者是实际内存的大小，Kbytes字节。

```
dell@r740:~$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0  78400  8628 ?        Ss   Oct27  52:32 /sbin/init maybe-ubiquity
root         2  0.0  0.0      0     0 ?        S    Oct27   0:00 [kthreadd]
root         4  0.0  0.0      0     0 ?        I<   Oct27   0:00 [kworker/0:0H]
```

两种置换策略　全局置换　局部置换



- 全局置换: 实现简单
- 但全局置换不能实现公平、保护: 一个经过巧妙优化的程序里会出现大量goto，则…

页面置换需要考虑的关键问题

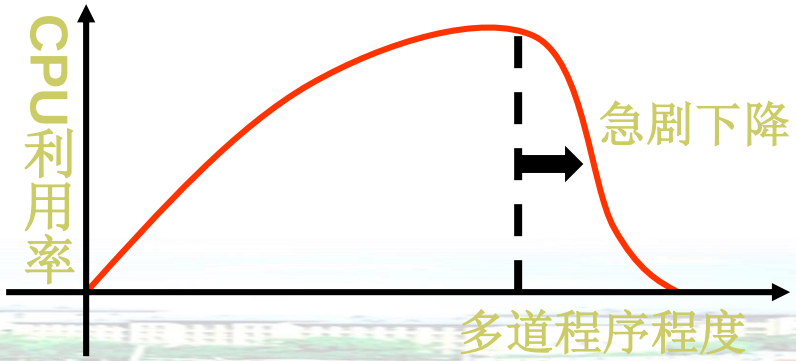
⑩ 给进程分配多少页框(帧frame)

- 分配的多，请求调页的意义就没了！一定要少？
- 至少是多少？可执行任意一条指令，如mov [a], [b]
- 是不是就选该下界值？

最坏情况需要6帧!

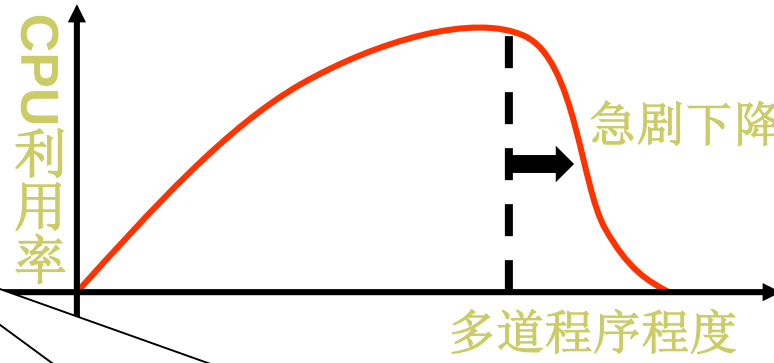
- 来看一个实例: 操作系统监视CPU使用率，发现CPU使用率太低时，向系统载入新进程。

会发生什么？



CPU利用率急剧下降的原因

- 系统内进程增多 ⇒ 每个进程的缺页率增大 ⇒ 缺页率增大到一定程度，进程总等待调页完成 ⇒ CPU利用率降低 ⇒ 进程进一步增多，缺页率更大 …
- 称这一现象为颠簸(thrashing)
- 显然，防止的根本手段给进程分配足够多的帧

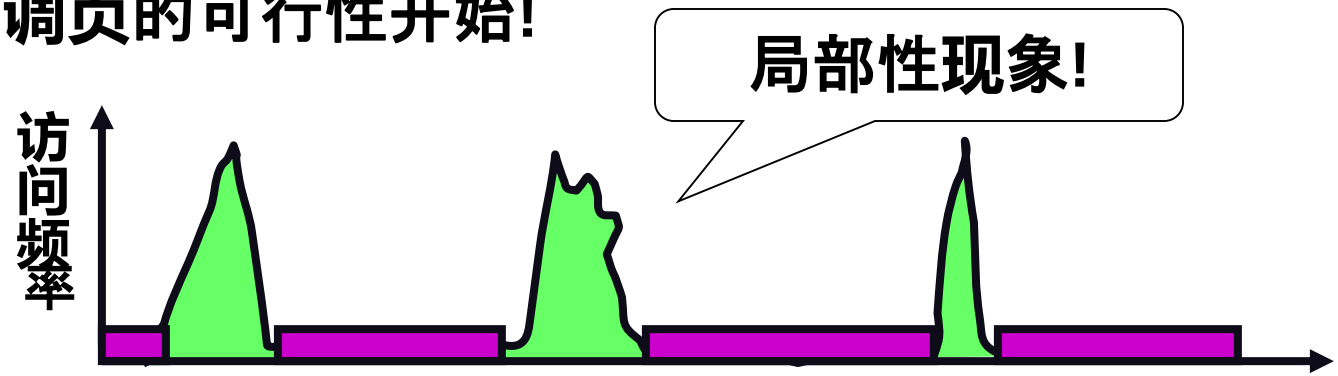


此时: 进程调入一页，需将一页淘汰出去，刚淘汰出去的页马上要需要调入，就这样……

问题是怎么确定进程需要多少帧才能不颠簸?

工作集模型

- 任何计算都需要一个模型! 要确定进程所需的帧数该依靠什么信息呢?
  - 从请求调页的可行性开始!

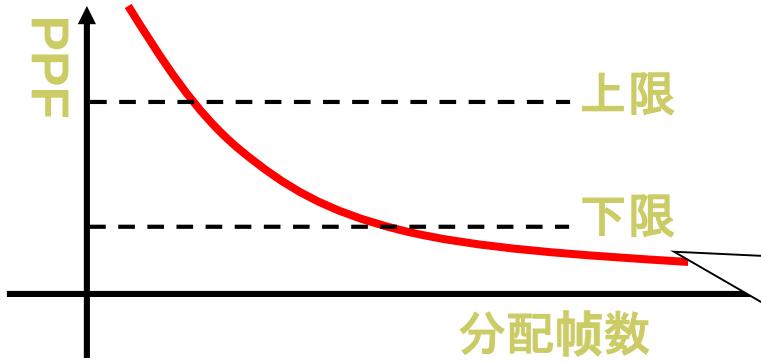


  - 只要分配的帧空间能覆盖整个局部就不会出现太多的缺页!
  - 工作集模型就用来计算一个局部的宽度(帧数)

基于页错误率的帧分配

- 页错误率(PFF) = 页错误/指令执行条数
  - 如果PFF>上限，增加分配帧数

如果没有空闲帧，则换出进程



有趣的是，帧数越多，PPF并不一定下降

  - 此种方法简单直接，在处理颠簸时常用。

那工作集呢？

  - 往往是PFF方法和工作集互相配合

  - 但现代OS并不十分重视颠簸现象，因为CPU更快了，进程很快exit；内存更大了，局部的变化不大

Belady异常

- 来看一个例子!
  - 引用序列：1，2，3，4，1，2，5，1，2，3，4，5

FIFO页置换

3frame

9faults

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

4frame

10faults

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

- **Belady异常现象：**对有的页面置换算法，页错误率可能会随着分配帧数增加而增加。

什么样的页置换没有Belady异常

看个模型!

m是分配的帧数

引用序列：1，2，3，4，1，2，5，1，2，3，4，5

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

m=3

m=4

LRU栈实现

- 特征: M(m, r)⊆M(m+1,r)，如{5,2,1}⊆{5,2,1,4} (m=3)

满足这一特征的算法称为栈式算法!

看看FIFO

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

m=3

m=4

- 结论：栈式算法无Belady异常，LRU属于栈式算法!

虚拟内存中程序优化

虚拟内存：按需调页与页面置换。

如何优化提升程序的性能？

对代码来说，紧凑的代码也往往意味着接下来执行的代码更大可能就在相同的页或相邻页。根据时间locality特性，程序90%的时间花在了10%的代码上。如果能将这10%的代码尽量紧凑且排在一起，无疑会大大提高程序的整体运行性能。

对数据来说，尽量将那些会一起访问的数据（比如链表）放在一起。这样当访问这些数据时，因为它们在同一页或相邻页，只需要一次调页操作即可完成；反之，如果这些数据分散在多个页（更糟的情况是这些页还不相邻），那么每次对这些数据的整体访问都会引发大量的缺页错误，从而降低性能。

整理一下前面的学习

❿ 虚拟内存的基本思想

- 将进程的一部分(不是全部)放进内存
- 其他部分放在磁盘
- 需要的时候调入: 请求调页

what?

内存利用率高，程序编制容易，响应时间快...

why?

❿ 请求调页的基本思想

- 当MMU发现页不在内存时，中断CPU
- CPU处理此中断，找到一个空闲页框
- CPU将磁盘上的页读入到该页框
- 如果没有空闲页框需要置换某页(LRU)

how?

虚拟内存总结

- 内存的根本目的 ⇒ 把程序放在内存并让其执行
- 只要将部分程序放进内存即可执行 ⇒ 内存利用率高
- 可编写比内存大的程序 ⇒ 使用一个大地址空间(虚拟内存)
- 部分程序在内存 ⇒ 其他部分在磁盘 ⇒ 需要的时候调入内存
- 页表项存在P位 ⇒ 缺页产生中断 ⇒ 中断处理完成页面调入
- 调入页面需要一个空闲页框 ⇒ 如果没有空闲页框 ⇒ 置换
- 置换方法 ⇒ FIFO→MIN→LRU→Clock（NRU）
- 需要给进程分配页框 ⇒ 全局、局部 ⇒ 颠簸 ⇒ 工作集

虚拟内存核心：段页管理 部分加载 按需调页 换入换出

CSAPP<虚拟内存>串讲

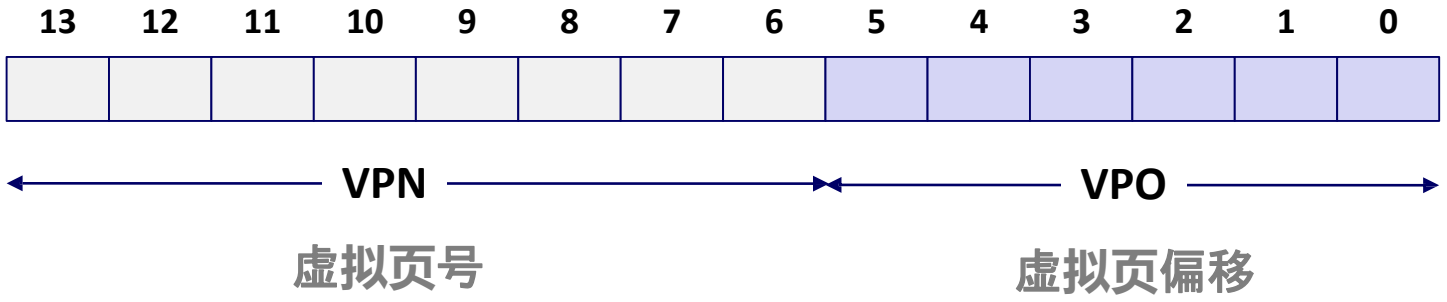
- 一个小内存系统示例
- 案例研究: Core i7/Linux 内存系统
- 内存映射

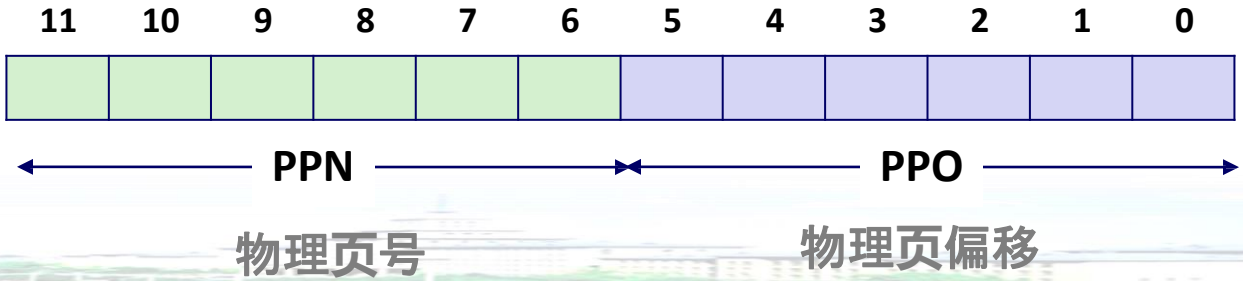
Review of Symbols符号回顾

- 基本参数
  - $N = 2^n$:虚拟地址空间中的地址数量
  - M = 2m : 物理地址空间中的地址数量
  - $P = 2^p$: 页的大小 (bytes)
- 虚拟地址组成部分
  - TLBI: TLB索引
  - TLBT: TLB 标记
  - VPO: 虚拟页面偏移量(字节)
  - VPN: 虚拟页号
- 物理地址组成部分
  - PPO:物理页面偏移量 (same as VPO)
  - PPN:物理页号
  - CO: 缓冲块内的字节偏移量
  - CI: Cache 索引
  - CT: Cache 标记

一个小内存系统示例

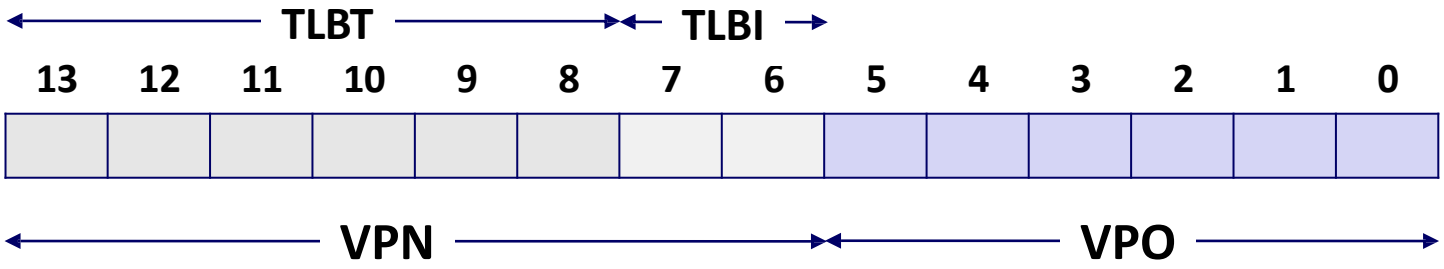
- **地址假设**
  - 14位虚拟地址（n=14）
  - 12位物理地址（m = 12）
  - 页面大小64字节（P = 64）





1. 小内存系统的 TLB

- 16 entries　16个条目
- 4-way associative 4路组相联



组	标记	PPN	有效位	标记	PPN	有效位	标记	PPN	有效位	标记	PPN	有效位
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

2. 小内存系统的页表

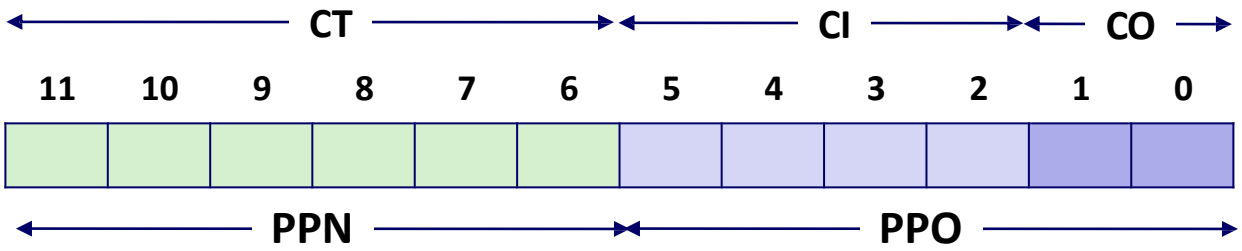
只展示了前16个PTE (out of 256)

VPN	PPN	有效位
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	有效位
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

3. 小内存系统的 Cache

- 16个组，每块为4字节
- 通过物理地址中的字段寻址
- 直接映射

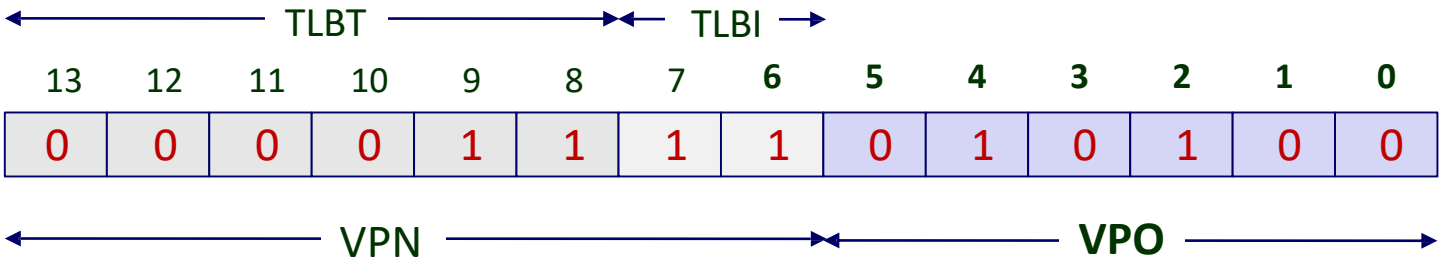


索引	标记位	有效位	块[0]	块[1]	块[2]	块[3]
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

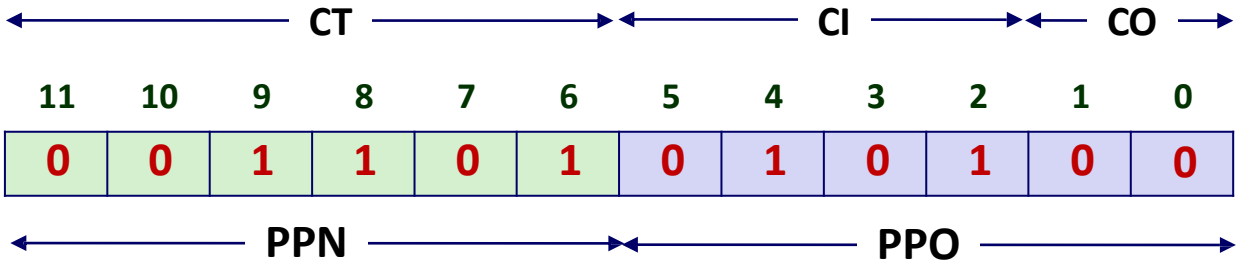
地址翻译 Example #1

虚拟地址: 0x03D4



VPN 0x0F　　TLBI 0x3　　TLBT 0x03　　TLB Hit? Y　　Page Fault? N　　PPN: 0x0D

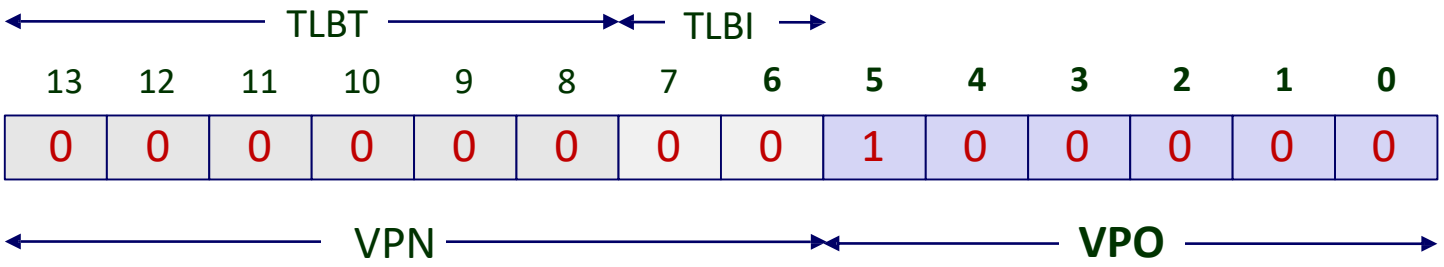
物理地址



CO 0　　CI 0x5　　CT 0x0D　　Hit? Y　　Byte: 0x36

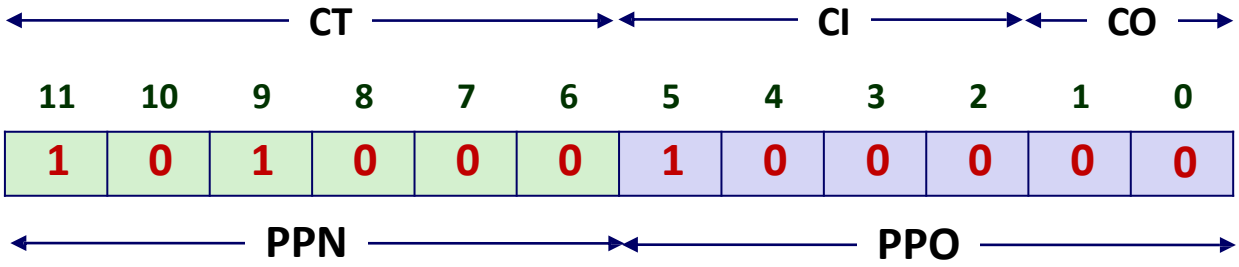
地址翻译 Example #2

虚拟地址: 0x0020



VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

物理地址

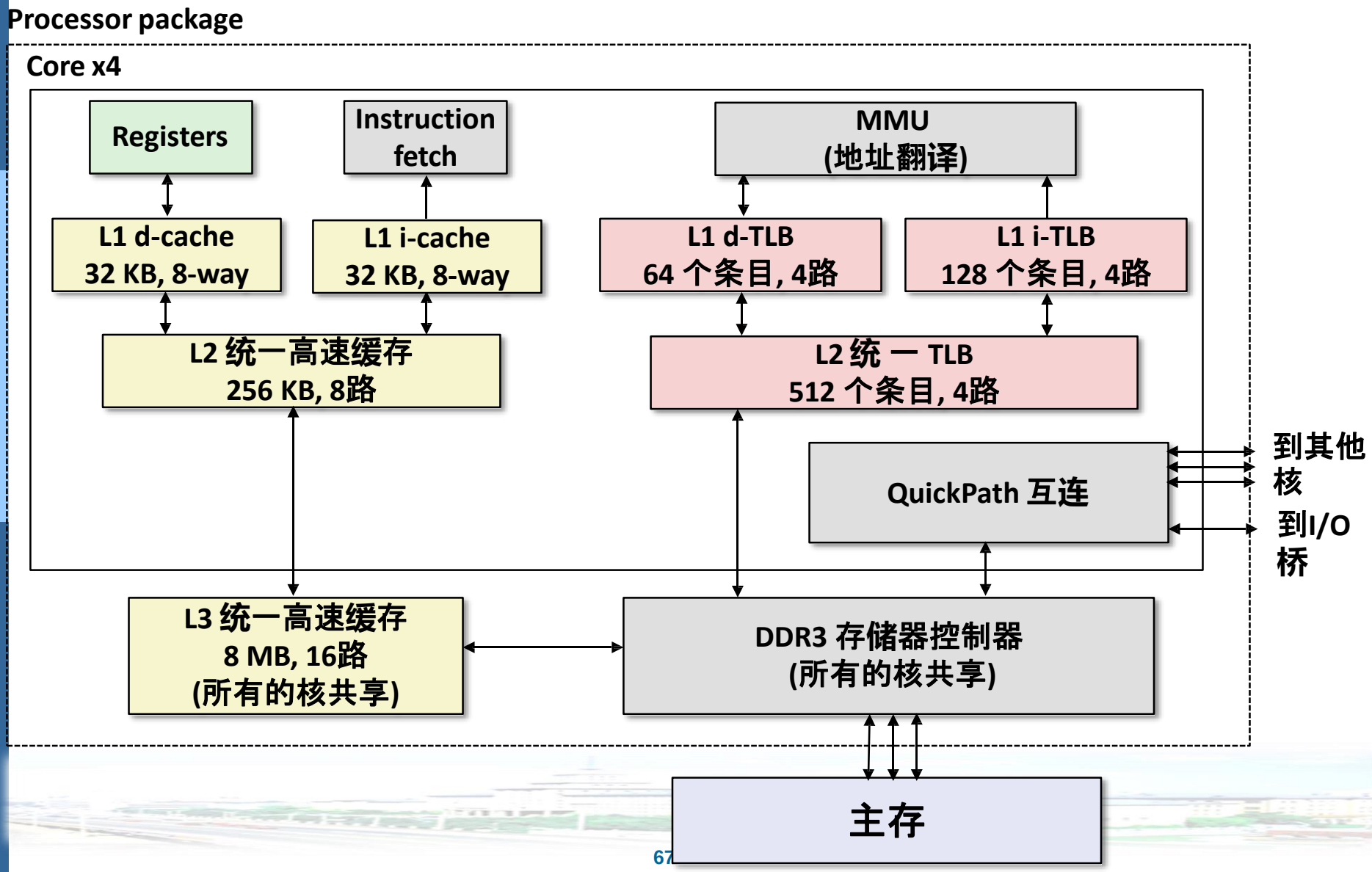


CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

CSAPP<虚拟内存>串讲

- 一个小内存系统示例
- **案例研究: Core i7/Linux 内存系统**
- 内存映射

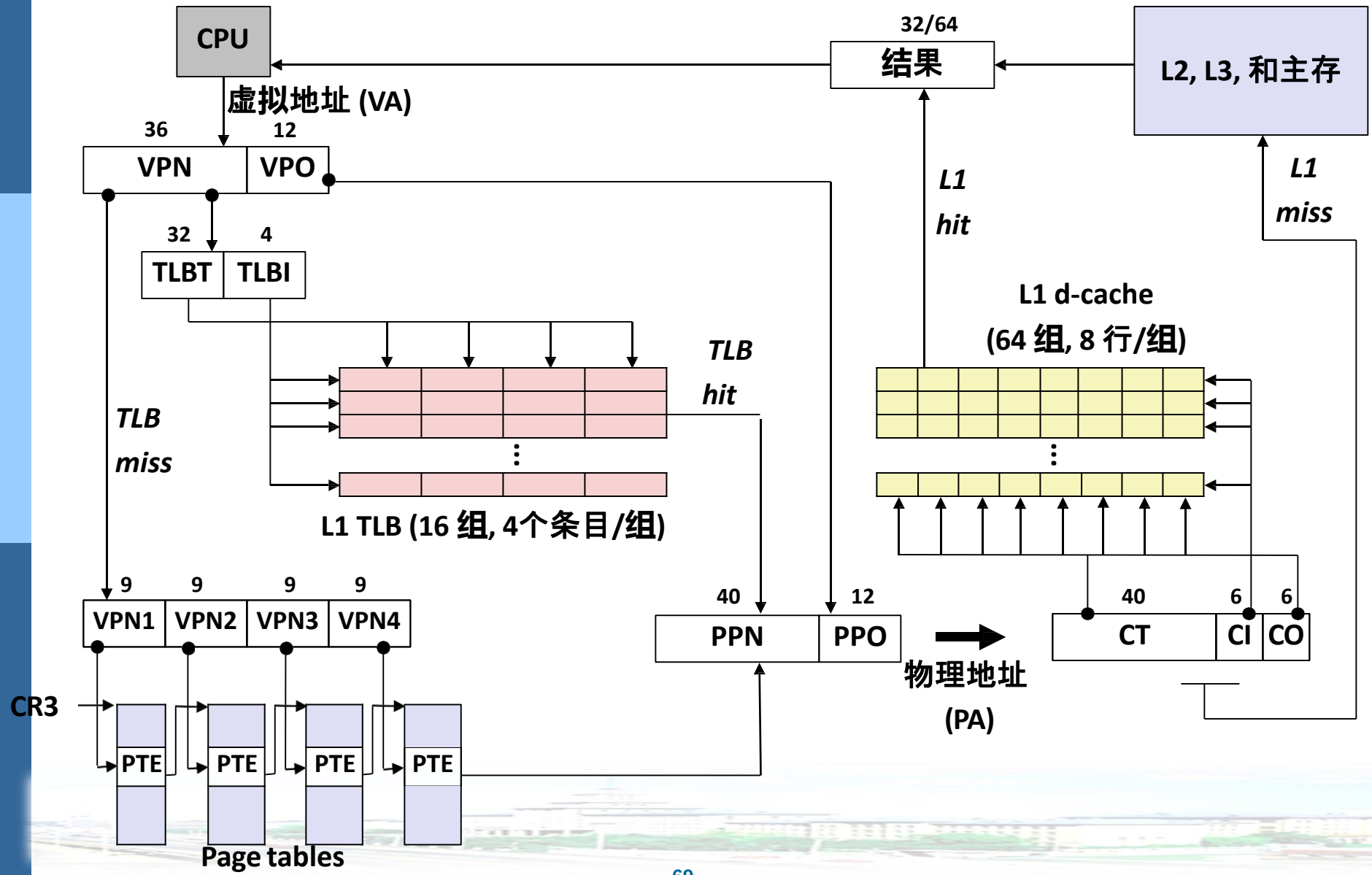
Intel Core i7 内存系统



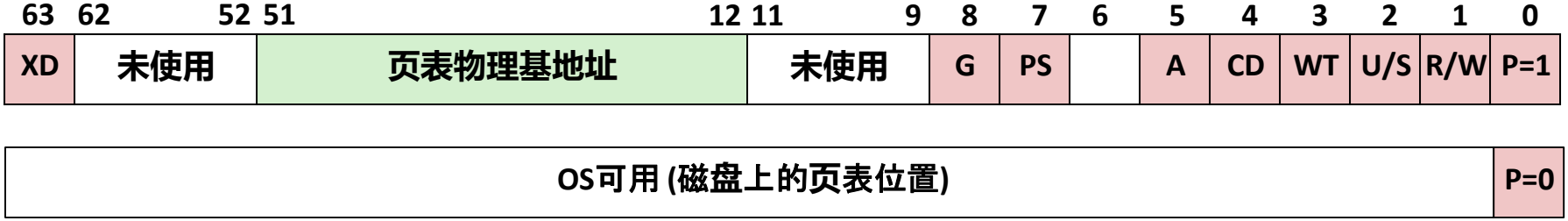
Review of Symbols符号回顾

- 基本参数
  - $N = 2^n$:虚拟地址空间中的地址数量
  - M = 2m : 物理地址空间中的地址数量
  - $P = 2^p$: 页的大小 (bytes)
- 虚拟地址组成部分
  - TLBI: TLB索引
  - TLBT: TLB 标记
  - VPO: 虚拟页面偏移量(字节)
  - VPN: 虚拟页号
- 物理地址组成部分
  - PPO:物理页面偏移量 (same as VPO)
  - PPN:物理页号
  - CO: 缓冲块内的字节偏移量
  - CI: Cache 索引
  - CT: Cache 标记

Core i7 地址翻译


CPU
32/64
结果
L2, L3, 和主存
虚拟地址 (VA)
36
12
VPN
VPO
L1
hit
L1
miss
32
4
TLBT
TLBI
L1 d-cache
(64 组, 8 行/组)
TLB
hit
TLB
miss
L1 TLB (16 组, 4个条目/组)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
40
12
PPN
PPO
物理地址
(PA)
40
6
6
CT
CI
CO
CR3
PTE
PTE
PTE
PTE
Page tables

Core i7 1-3级页表条目格式



每个条目引用一个 4KB子页表:

P: 子页表在物理内存中 (1)不在 (0).

R/W: 对于所有可访问页，只读或者读写访问权限.

U/S: 对于所有可访问页，用户或超级用户 (内核)模式访问权限.

WT: 子页表的直写或写回缓存策略.

A: 引用位 (由MMU 在读或写时设置，由软件清除).

PS: 页大小为4 KB 或 4 MB (只对第一层PTE定义).

Page table physical base address: 子页表的物理基地址的最高40位 (强制页表 4KB 对齐)

XD: 能/不能从这个PTE可访问的所有页中取指令.

对照书 p578

Core i7 第 4 级页表条目格式

每个条目引用一个 4KB子页表:

P: 子页表在物理内存中 (1)不在 (0).

R/W: 对于所有可访问页，只读或者读写访问权限.

U/S: 对于所有可访问页，用户或超级用户 (内核)模式访问权限.

WT: 子页表的直写或写回缓存策略.

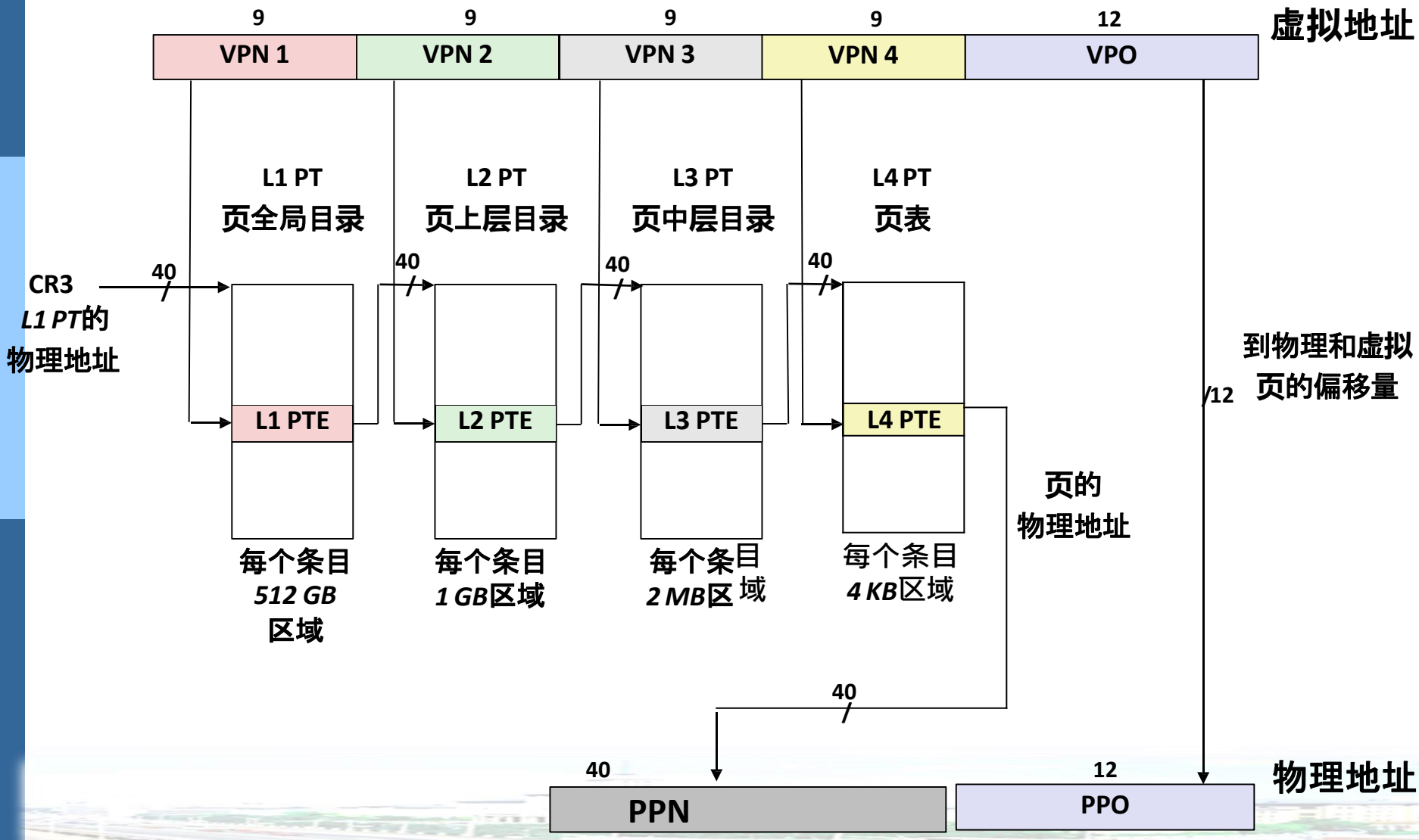
A:引用位 (由MMU 在读或写时设置，由软件清除).

D: 修改位 (由MMU 在读和写时设置，由软件清除)

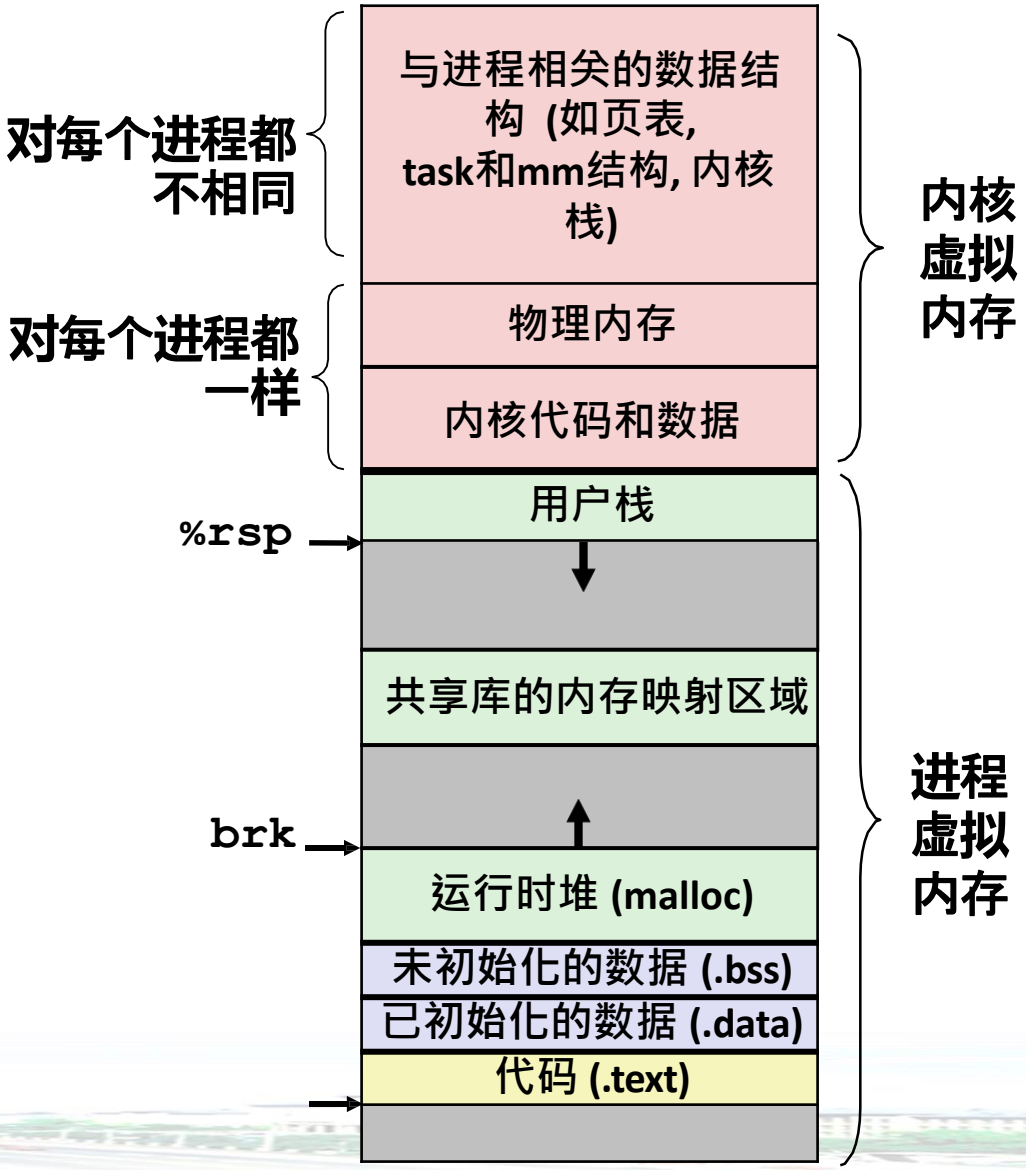
Page table physical base address: 子页表的物理基地址的最高40位 (强制页表 4KB 对齐)

XD: 能/不能从这个PTE可访问的所有页中取指令.

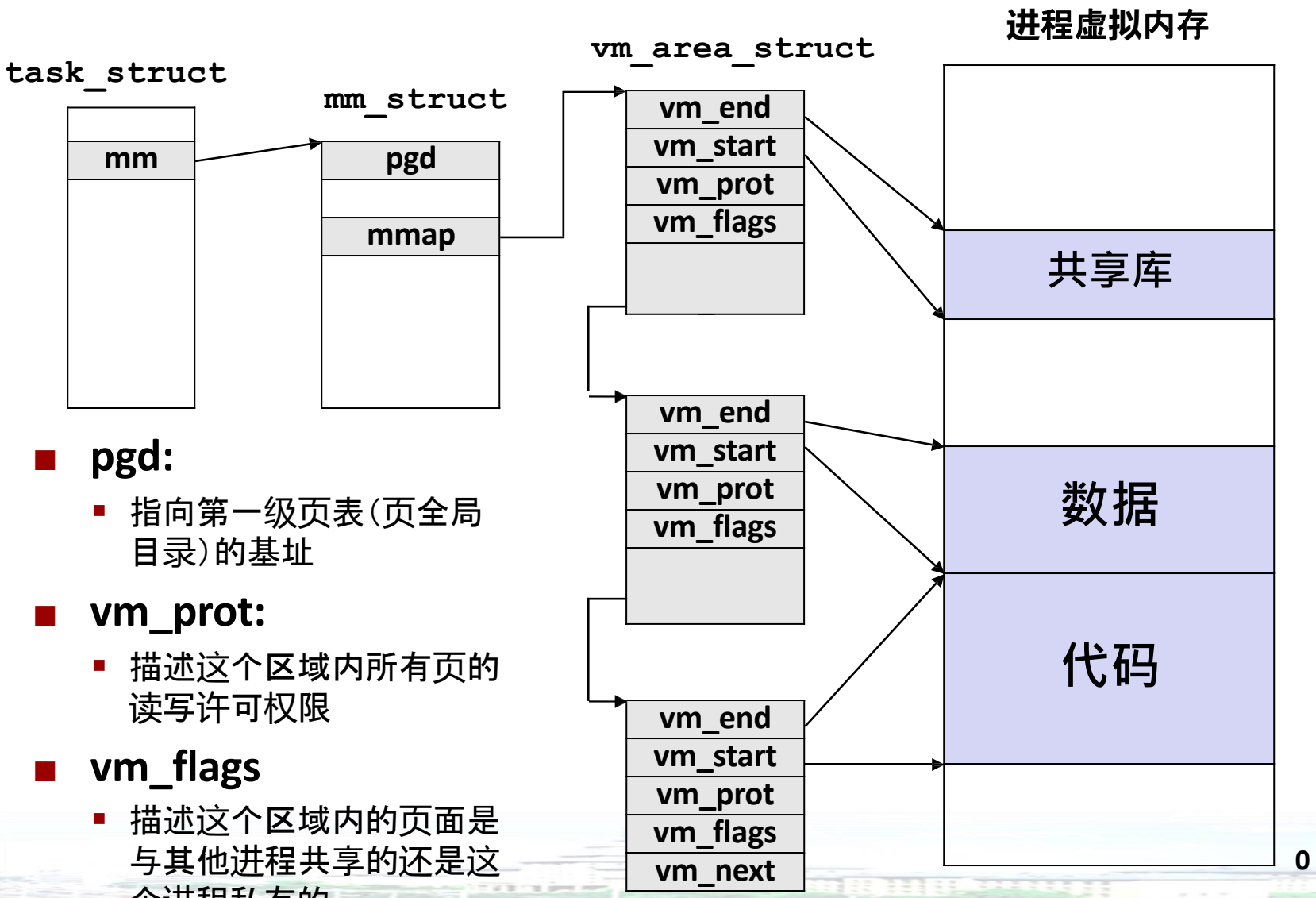
Core i7 页表翻译


9
9
9
9
12
虚拟地址
VPN 1
VPN 2
VPN 3
VPN 4
VPO
L1 PT
页全局目录
L2 PT
页上层目录
L3 PT
页中层目录
L4 PT
页表
CR3
L1 PT的
物理地址
40
40
40
40
L1 PTE
L2 PTE
L3 PTE
L4 PTE
到物理和虚拟
页的偏移量
12
页的
物理地址
每个条目
512 GB
区域
每个条目
1 GB区域
每个条目
2 MB区域
每个条目
4 KB区域
40
40
12
PPN
PPO
物理地址

一个Linux 进程的虚拟地址空间

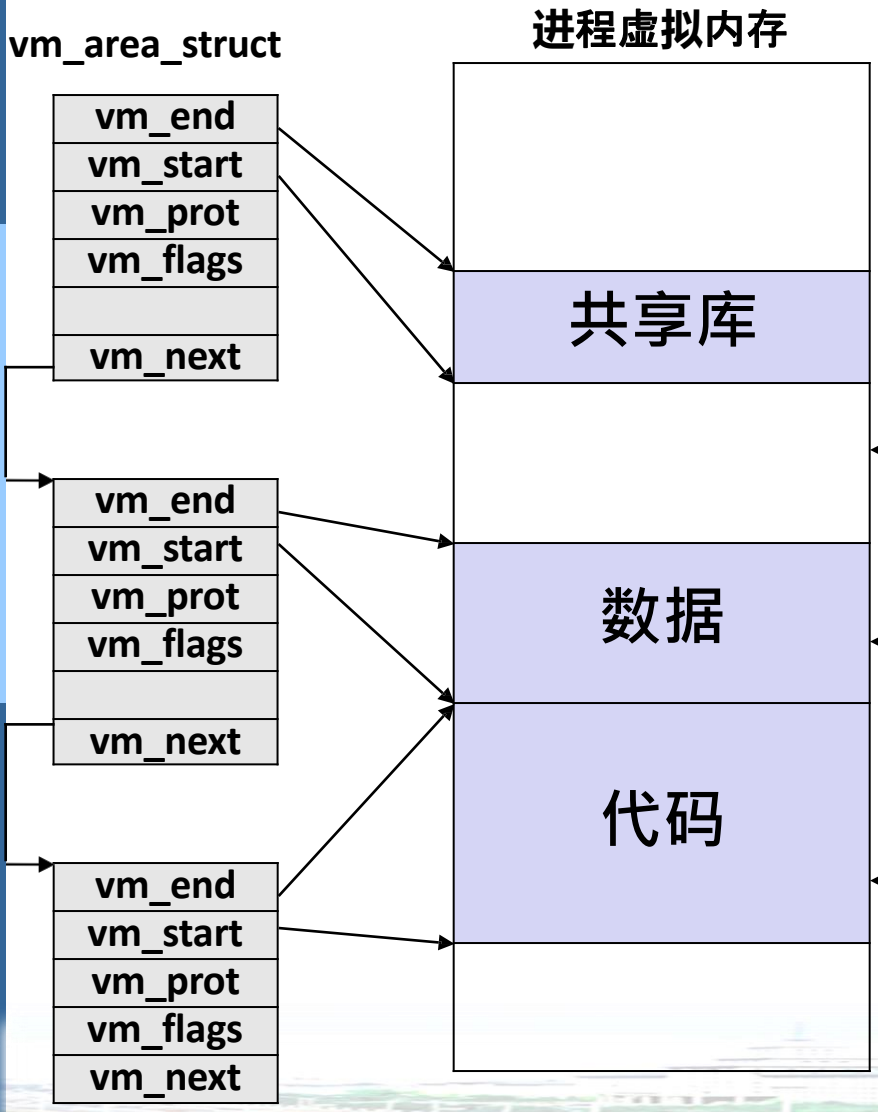


Linux将虚拟内存组织成一些区域的集合



- **pgd:**
  - 指向第一级页表(页全局目录)的基址
- **vm_prot:**
  - 描述这个区域内所有页的读写许可权限
- **vm_flags**
  - 描述这个区域内的页面是与其他进程共享的还是这个进程私有的

Linux缺页处理




```
Segmentation fault
/bin/sh: error while loading shared libraries: ▪ F8▪$▪ET▪▪
cannot open shared object file: No such file or directory
Segmentation fault
Segmentation fault
```

段错误:
访问一个不存在的页面

正常缺页

保护异常:
例如,违反许可，写一个只读的页面(Linux 报告 Segmentation fault)

CSAPP<虚拟内存>串讲

- 一个小内存系统示例
- 案例研究: Core i7/Linux 内存系统
- **内存映射**

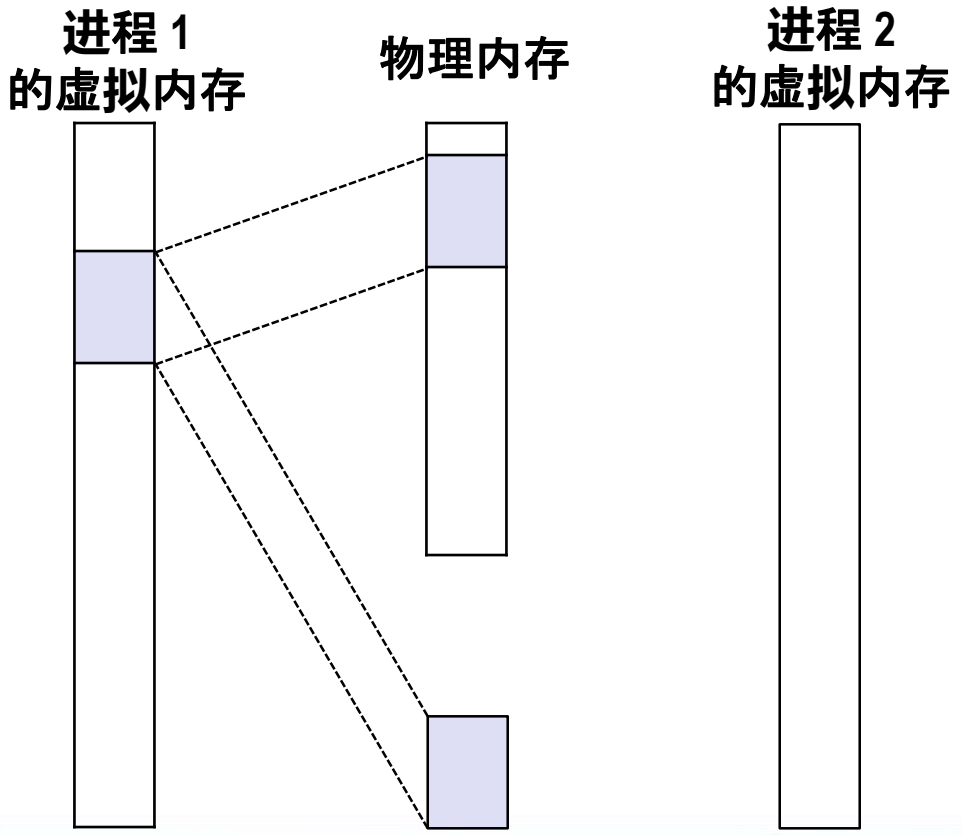
内存映射

- **Linux通过将虚拟内存区域与磁盘上的对象关联起来以初始化这个虚拟内存区域的内容.**
  - 这个过程称为内存映射(***memory mapping***).

- **虚拟内存区域可以映射的对象 (根据初始值的不同来源分) :**
  - **磁盘上的普通文件** (e.g.,一个可执行目标文件)
    - 文件区被分成页大小的片, 对虚拟页面初始化(执行按需页面调度)
  - **匿名文件** (内核创建, 包含的全是二进制零)
    - CPU第一次引用该区域内的虚拟页面时会分配一个全是零的物理页 (***demand-zero page*请求二进制零的页**)
    - 一旦该页面被修改, 即和其他页面一样
- **初始化后的页面在内存和交换文件(*swap file*)之间换来换去**

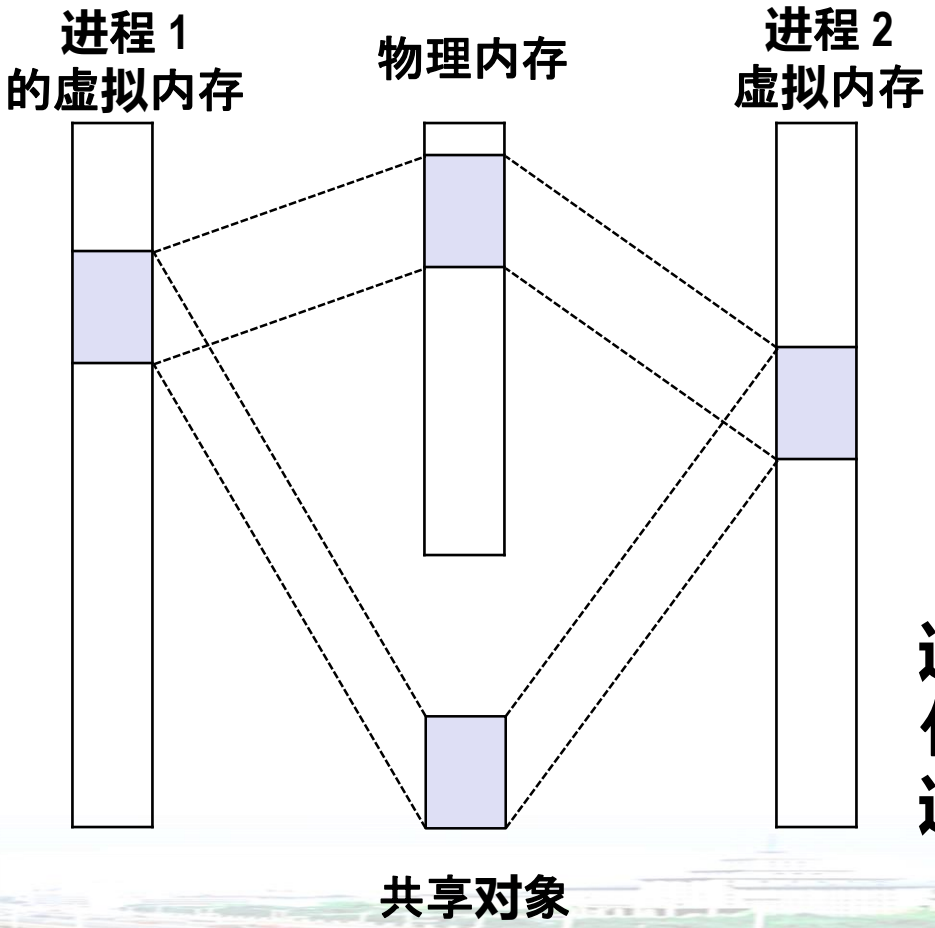
再看共享对象

一个对象被映射到虚拟内存的一个区域，要么作为共享对象，要么作为私有对象



- 进程 1 映射了一个共享对象

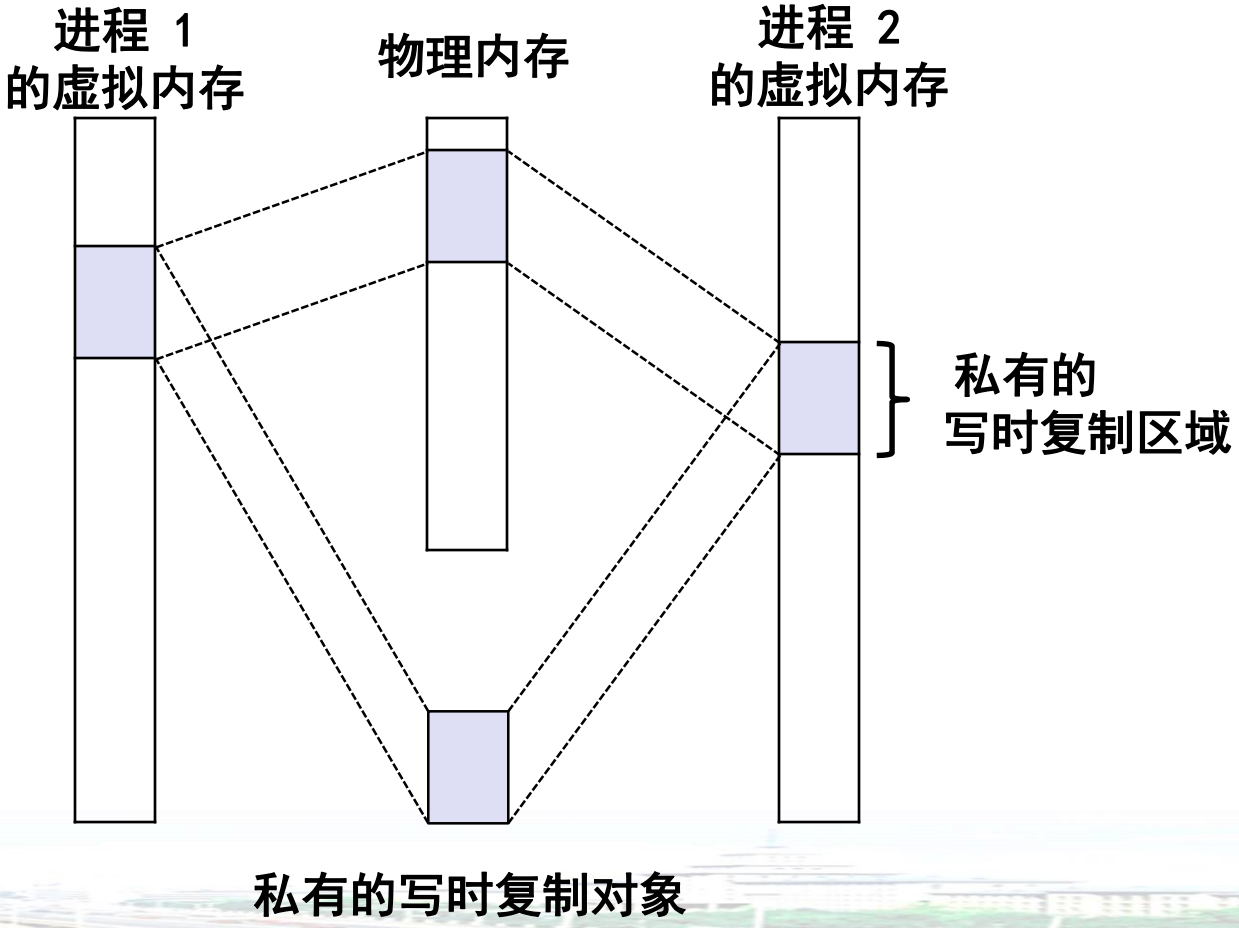
再看共享对象



- 进程 2 映射了同一个共享对象.
- 两个进程的虚拟地址可以是不同的
- 物理内存中只有一个该共享对象的副本

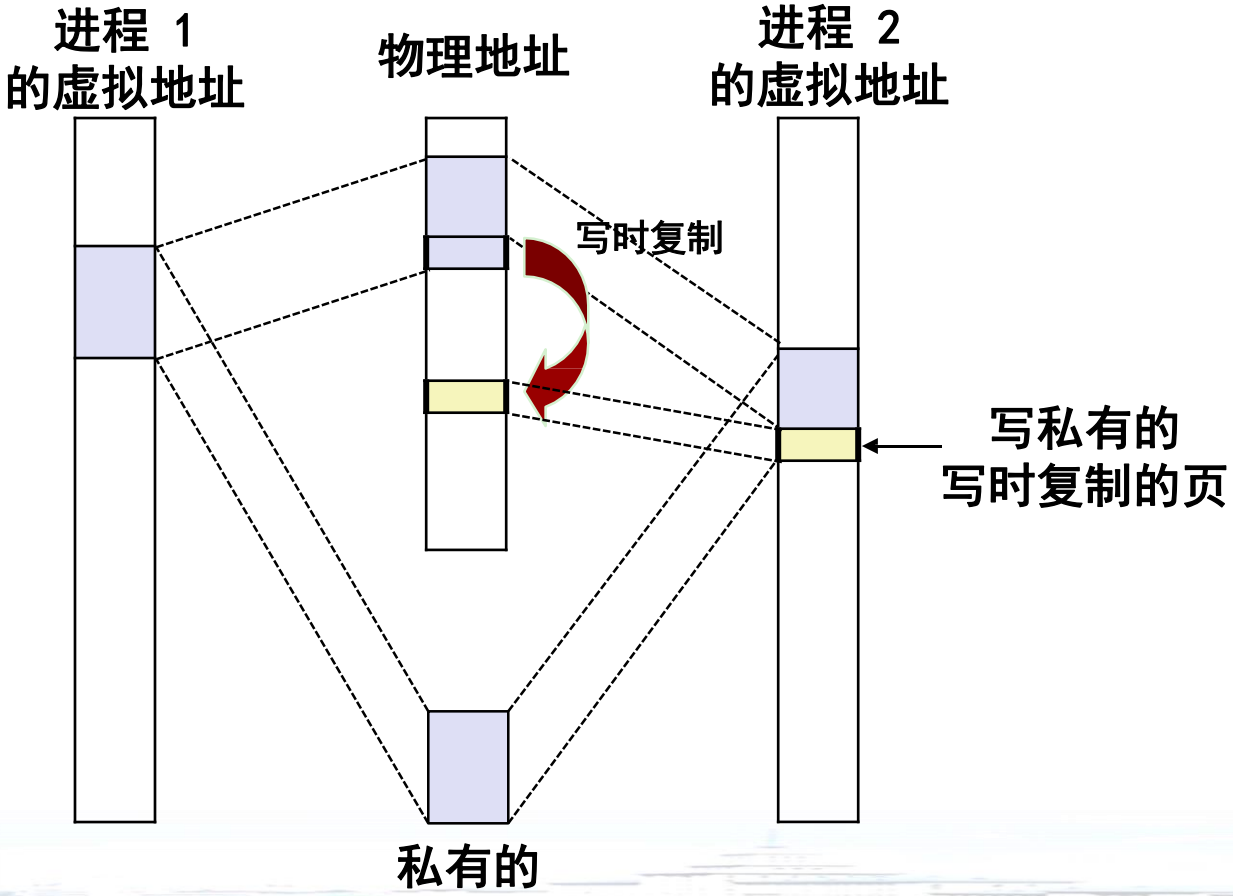
进程1对共享对象的人任何写操作(在进程1的虚拟内存区域中进行)对进程2都是可见的

私有的写时复制（Copy-on-write）对象



- 两个进程都映射了私有的写时复制对象
- 区域结构被标记为私有的写时复制
- 私有区域的页表条目都被标记为只读

私有的写时复制（Copy-on-write）对象



- 写私有页的指令触发保护故障
- 故障处理程序创建这个页面的一个新副本
- 故障处理程序返回时重新执行写指令
- 尽可能地延迟拷贝（创建副本）

再看 fork 函数

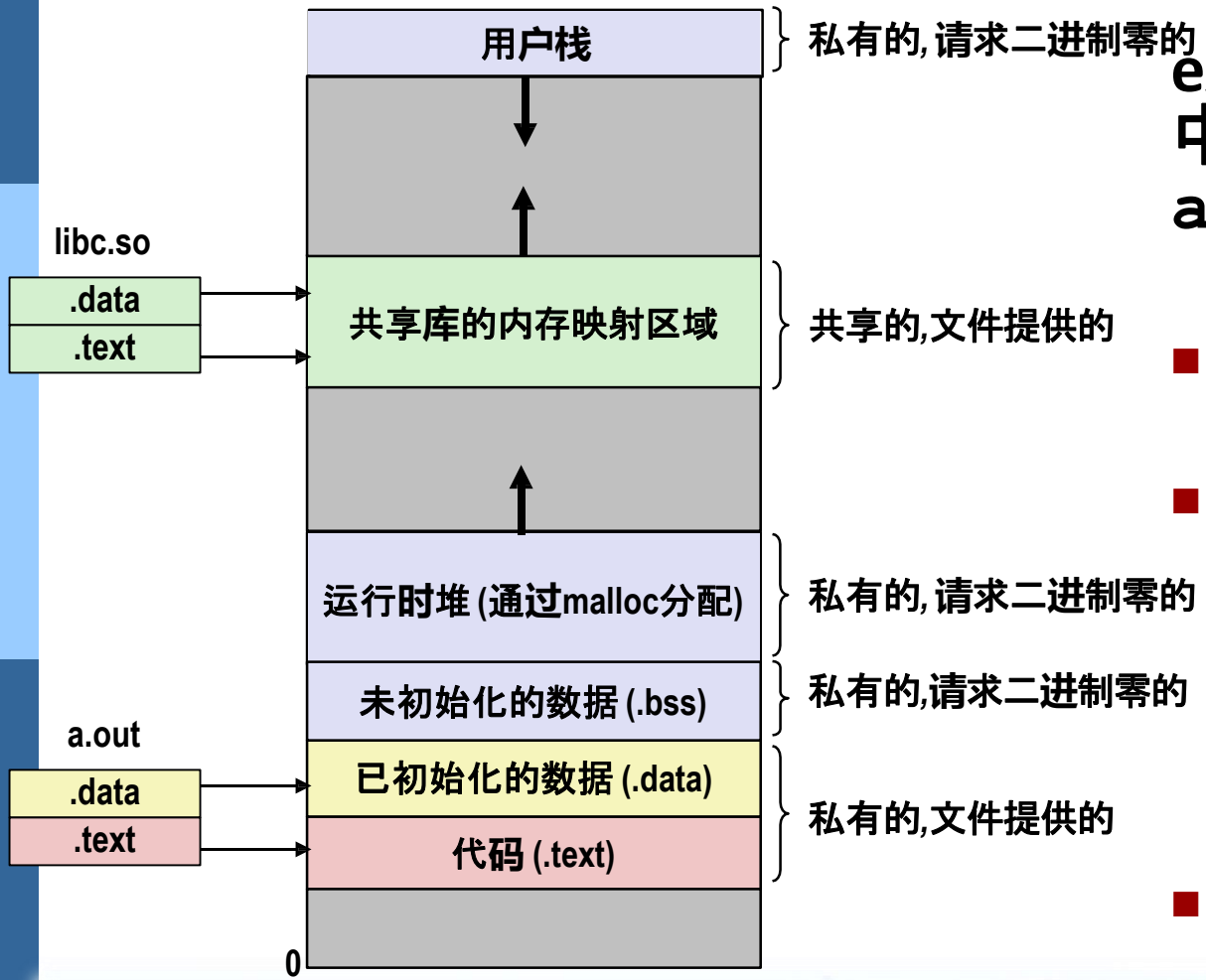
- 虚拟内存和内存映射解释了fork函数如何为每个新进程提供私有的虚拟地址空间.

- 为新进程创建虚拟内存
  - 创建当前进程的的mm_struct, vm_area_struct和页表的原样副本.
  - 两个进程中的每个页面都标记为只读
  - 两个进程中的每个区域结构（vm_area_struct）都标记为私有的 写时复制（COW）

- 在新进程中返回时，新进程拥有与调用fork进程相同的虚拟内存

随后的写操作通过写时复制机制创建新页面

再看 execve 函数



execve函数在当前进程中加载并运行新程序a.out的步骤:

- **删除已存在的用户区域**
- **创建新的区域结构**
  - 代码和初始化数据映射到.text和.data区(目标文件提供)
  - .bss和栈映射到匿名文件
- **设置PC，指向代码区域的入口点**
  - Linux根据需要换入代码和数据页面

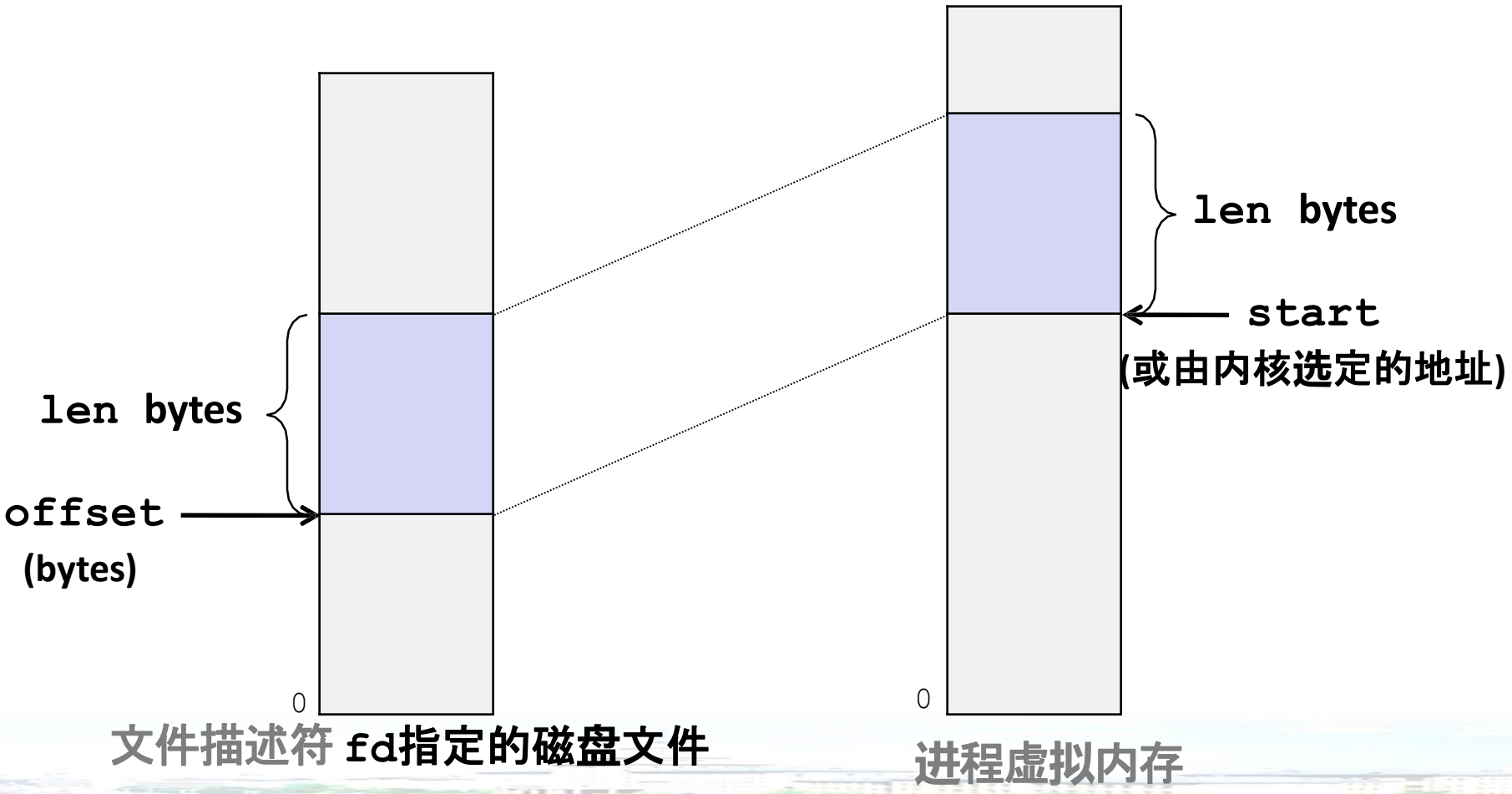
用户级内存映射

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- 从fd指定的磁盘文件的offset处映射len个字节到一个新创建的虚拟内存区域，该区域从地址start处开始
  - **start**: 虚拟内存的起始地址，通常定义为NULL
  - **prot**: 虚拟内存区域的访问权限，PROT_READ, PROT_WRITE, ...
  - **flags**: 被映射对象的类型，MAP_ANON（匿名对象），MAP_PRIVATE（私有的写时复制对象），MAP_SHARED（共享对象），...
- 返回一个指向映射区域开始处的指针

用户级内存映射

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Example: 使用 mmap 函数拷贝文件

- 拷贝一个文件到 stdout

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

  /* Ptr to memory mapped area */
    char *bufp;

    bufp = Mmap(NULL,size,
                PROT_READ,
                MAP_PRIVATE,
                fd, 0);
    Write(1, bufp, size);
    return;
}
```

标准输出

mmapcopy.c

```
/* mmapcopydriver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmdline arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
               argv[0]);
        exit(0);
    }
    /* Copy input file to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
```

mmapcopy.c

拓展阅读

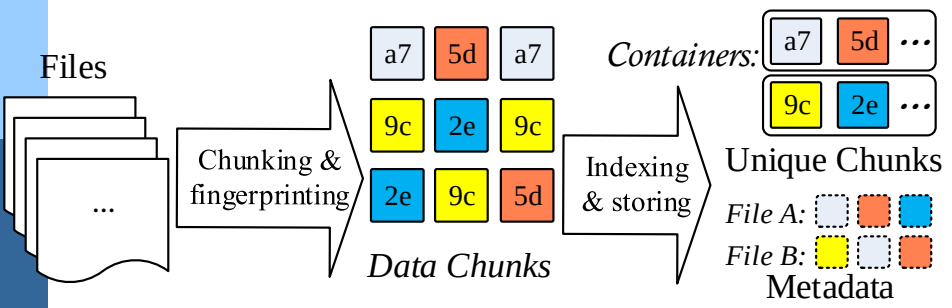
Avoiding the Disk Bottleneck in the Data Domain Deduplication File System

----FAST-08

背景：数据去重技术

数据去重（Data Deduplication）是本世纪初提出的一种文件级或者块级的压缩技术。数据去重与传统的压缩技术的本质区别是提出了数据块级或者是文件级粒度的冗余消除技术。

A.安全指纹标识算法　　B.基于内容分块算法

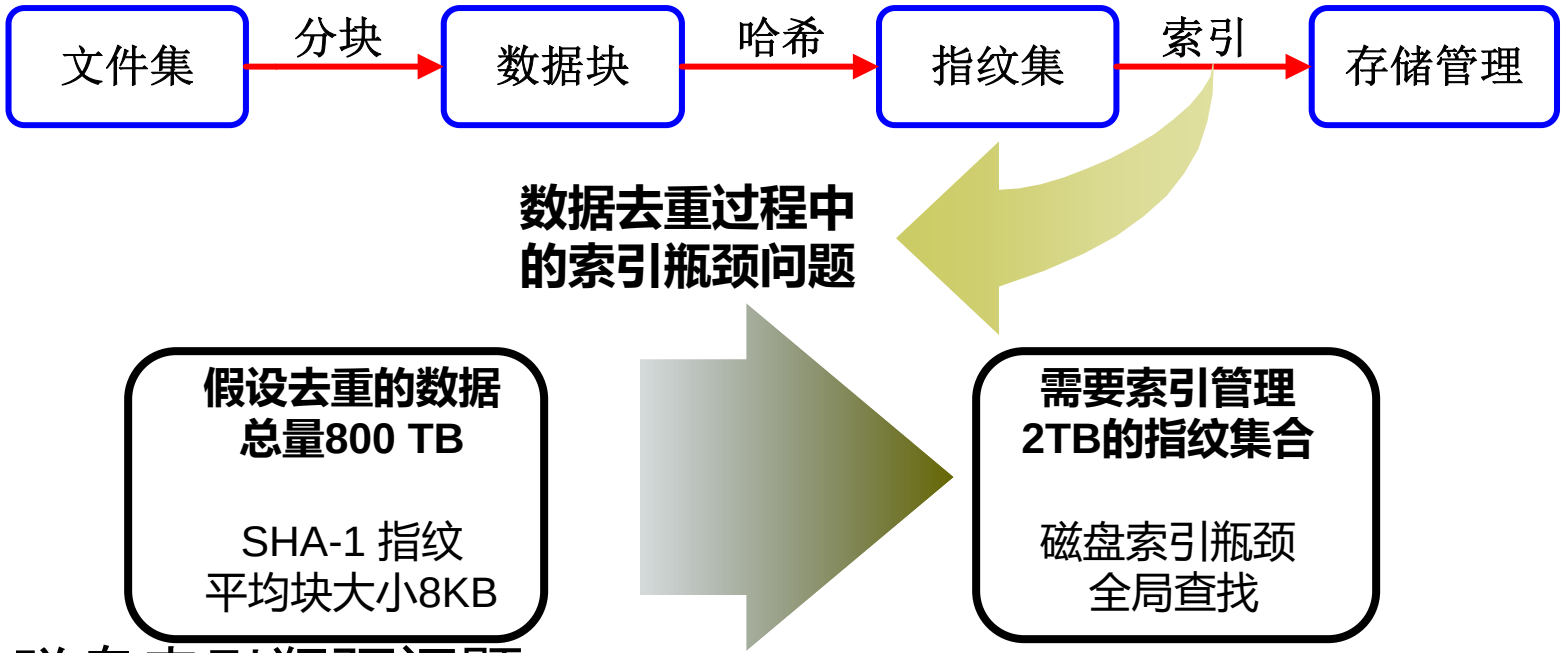


数据去重的基本流程：分块、哈希、索引和存储管理

提高了存储和传输的效率

- *极大地降低了存储空间的需求，从而减轻存储成本*
- *极大地减少了需要网络传送的数据量，从而提高传输带宽*

数据去重索引研究背景



磁盘索引瓶颈问题

Venti使用了磁盘指纹索引方案，其系统冗余数据消除吞吐率仅仅为6.5MB/s。

但是高吞吐率是企业数据保护的关键指标，其备份任务通常得超过100MB/s。

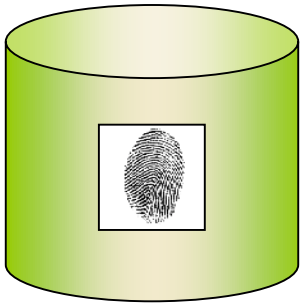
问题：指纹索引表

索引表大小（使用SHA1指纹）

8TB 数据, 20GB 指纹库

800TB 数据 ,2000GB 指纹库

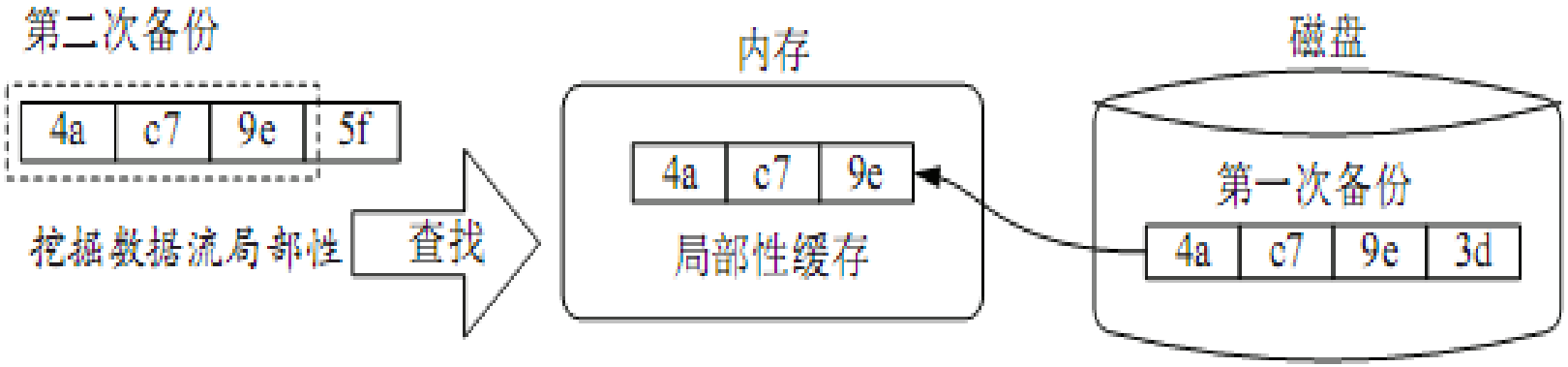
使用SHA256，SHA512呢？



太慢

太小

解决方案



内存开销（SHA1）？

1G个数据块→8TB数据→BF开销1GB

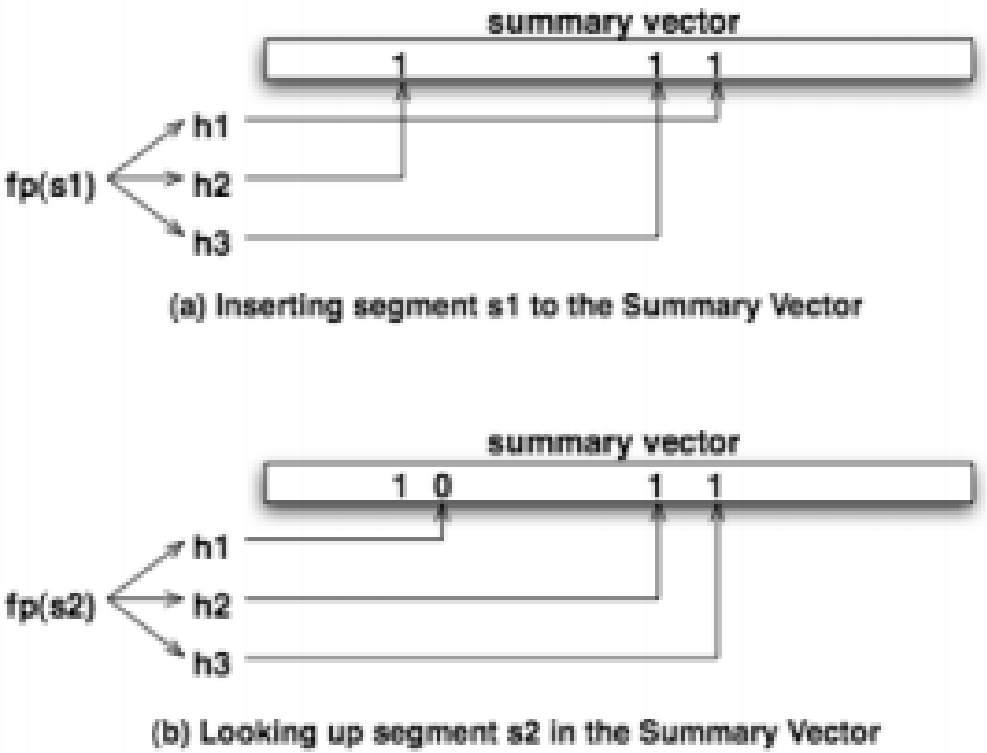
索引性能？

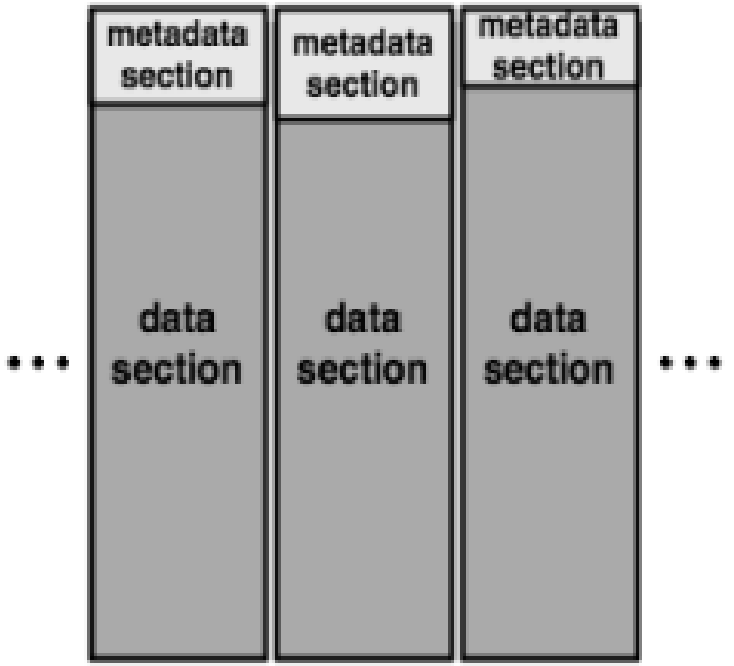
局部性+BF(布隆过滤器)

减少了99%的磁盘I/O

解决方案(续)

新型的索引结构:布隆过滤器





测试结果

Data center A backs up structured database data over the course of 31 days during the initial deployment of a deduplication system. The backup policy is to do daily full backups, where each full backup produces over 600 GB at steady state. There are two exceptions:

- During the initial seeding phase (until 6th day in this example), different data or different types of data are rolled into the backup set, as backup administrators figure out how they want to use the deduplication system. A low rate of duplicate segment identification and elimination is typically associated with the seeding phase.
- There are certain days (18th day in this example) when no backup is generated.

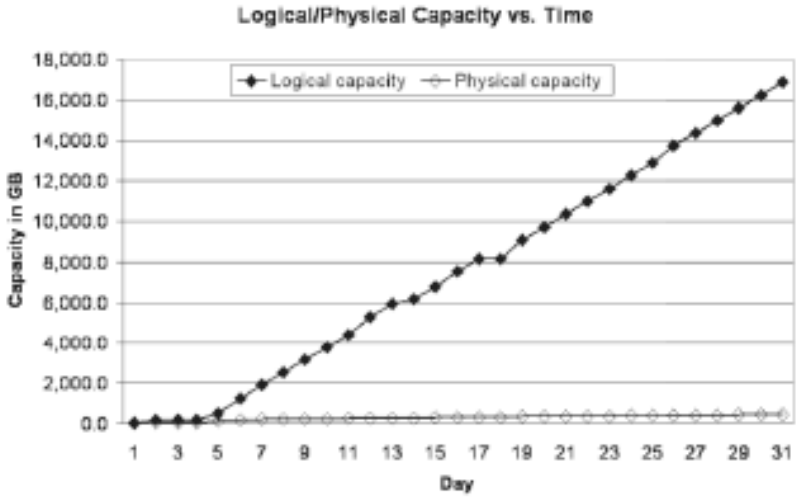


Figure 4: Logical/Physical Capacities at Data Center A

	Exchange data		Engineering data	
	# disk I/Os	% of total	# disk I/Os	% of total
no Summary Vector and no Locality Preserved Caching	328,613,503	100.00%	318,236,712	100.00%
Summary Vector only	274,364,788	83.49%	259,135,171	81.43%
Locality Preserved Caching only	57,725,844	17.57%	60,358,875	18.97%
Summary Vector and Locality Preserved Caching	3,477,129	1.06%	1,257,316	0.40%

影响力

全面取代磁带备份

带动了这个数据去重领域的研究

EMC收购，仍然是最大的备份产商之一