哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

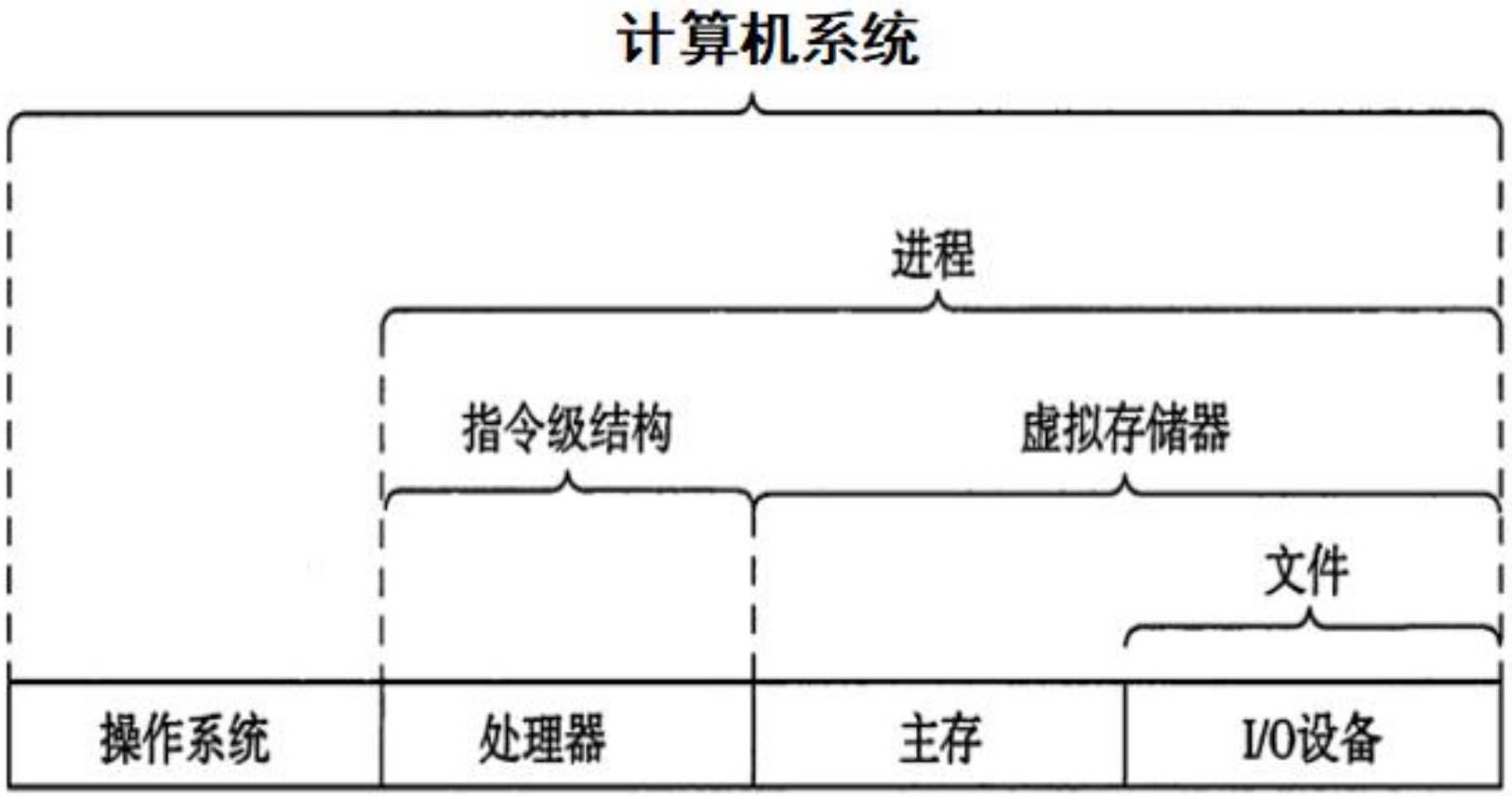
操作系统 (Operating System)

第七章：虚拟内存

夏文 副教授
哈尔滨工业大学（深圳）
2021年秋季

Email：xiawen@hit.edu.cn

计算机系统的抽象/虚拟化



1、导论与操作系统结构
2、进程与线程
3、并发与同步
4、处理器调度
5、死锁
6、内存管理
7、虚拟内存
8、I/O与存储
9、文件及文件系统

回顾：描述一下段页结合的内存地址翻译过程

段号+偏移(cs:ip)

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

逻辑地址

页号 | 偏移

常称为线性地址，以示区别

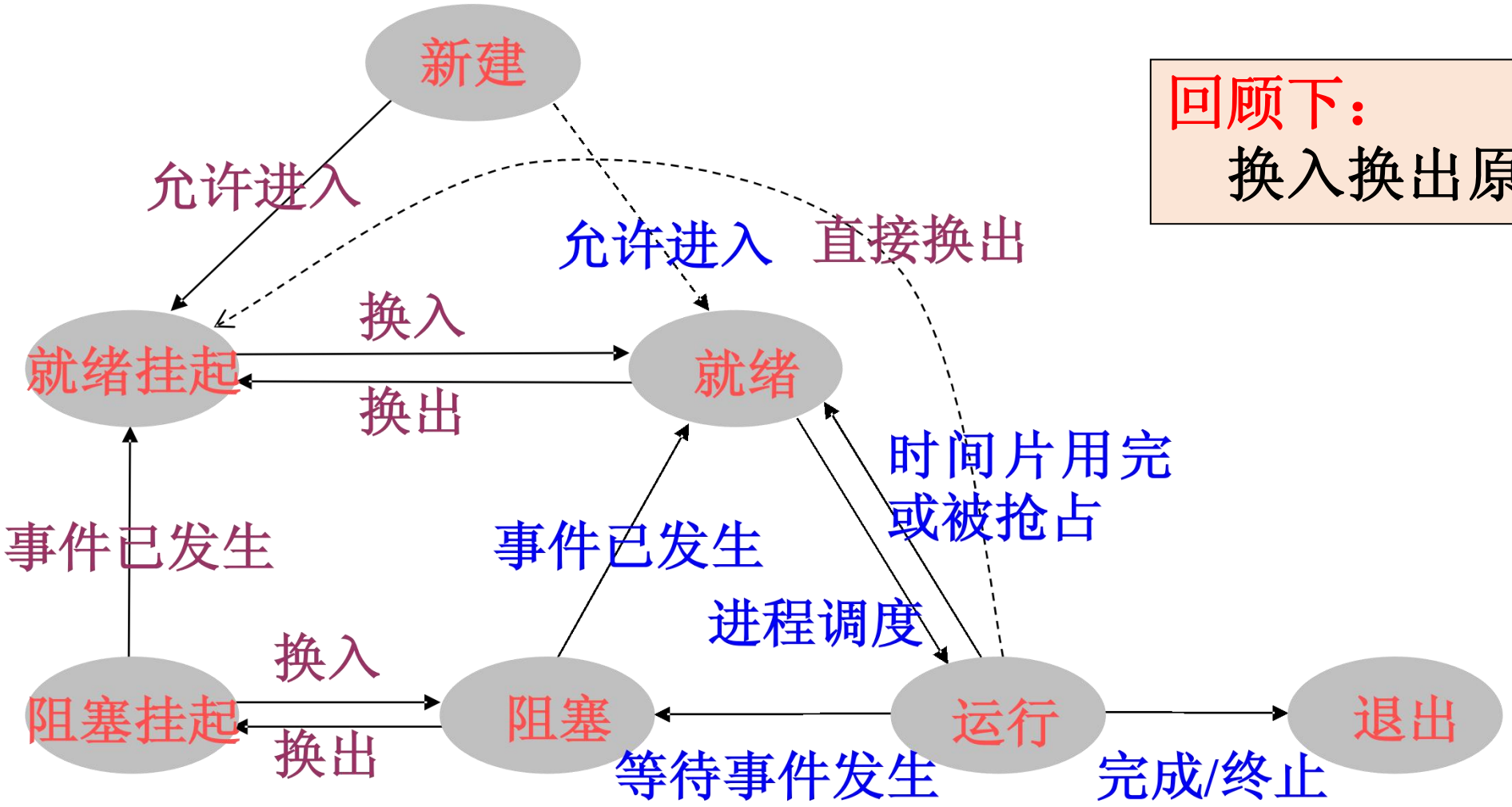
物理地址

物理页号 | 偏移

页框号	保护
5	R
1	R/W
3	R/W
7	R

思考：翻译逻辑地址0xc2f0、 0x0240 ？

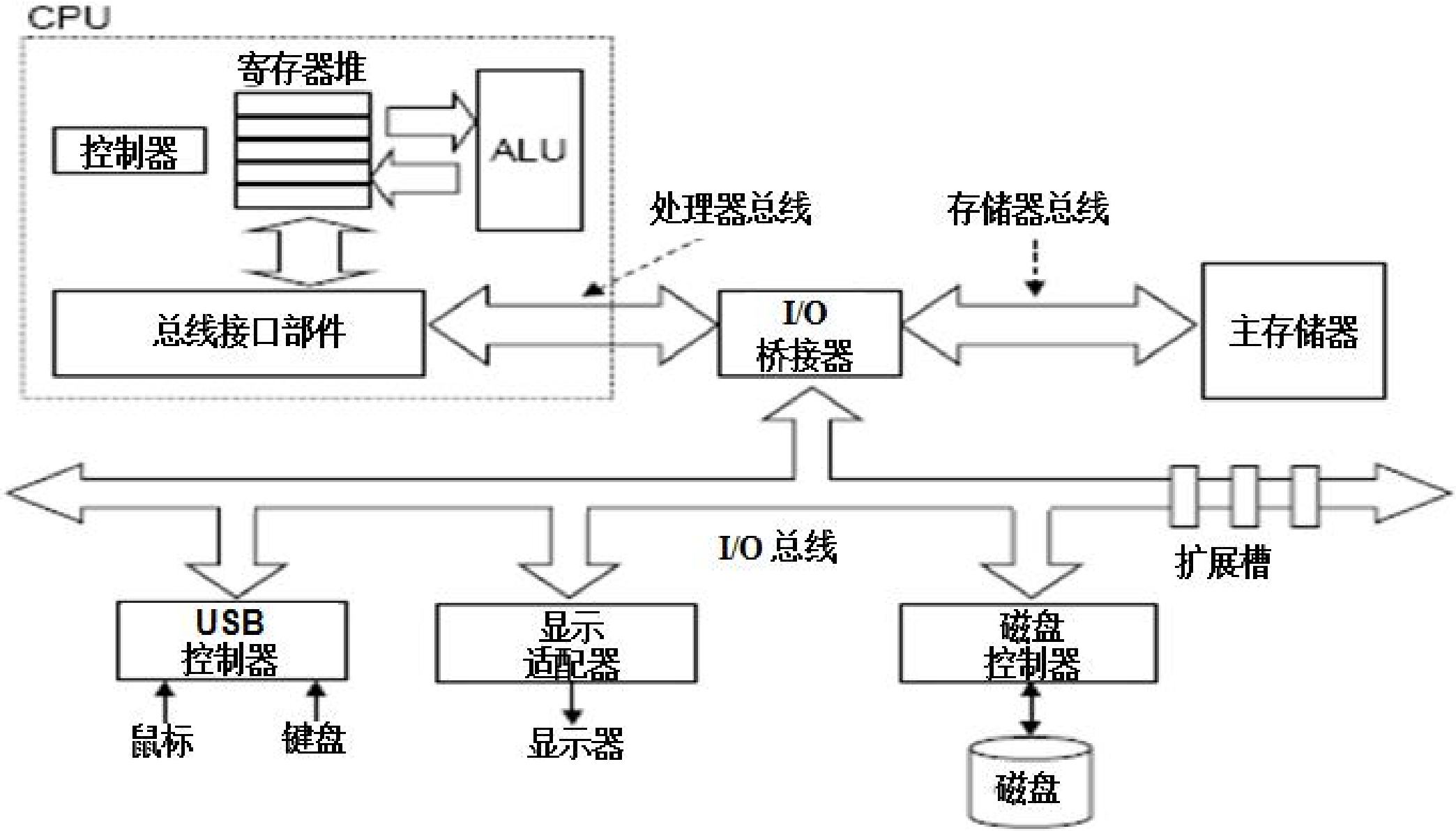
回顾：进程的描述与表达



回顾下：
换入换出原则？

挂起状态 – 引入主存←→外存的交换机制，虚拟存储管理的基础

计算机的体系结构


CPU
寄存器堆
控制器
ALU
处理器总线
存储器总线
总线接口部件
I/O
桥接器
主存储器
I/O 总线
扩展槽
USB
控制器
显示
适配器
磁盘
控制器
鼠标
键盘
显示器
磁盘

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

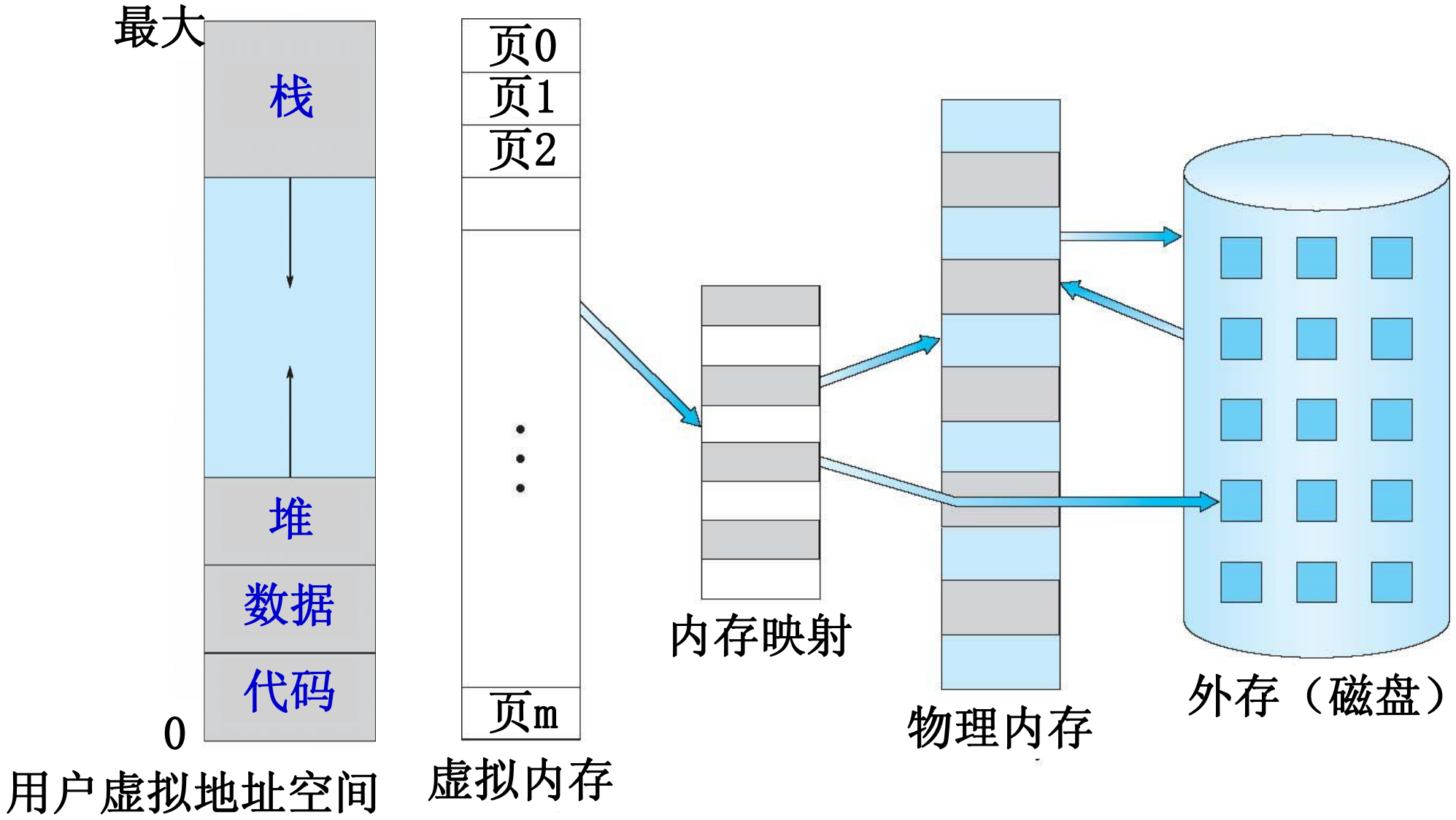
（4）系统颠簸现象和Belady异常现象 （5）虚拟内存中程序优化

- 7.5 CSAPP第9章<虚拟内存>串讲

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?



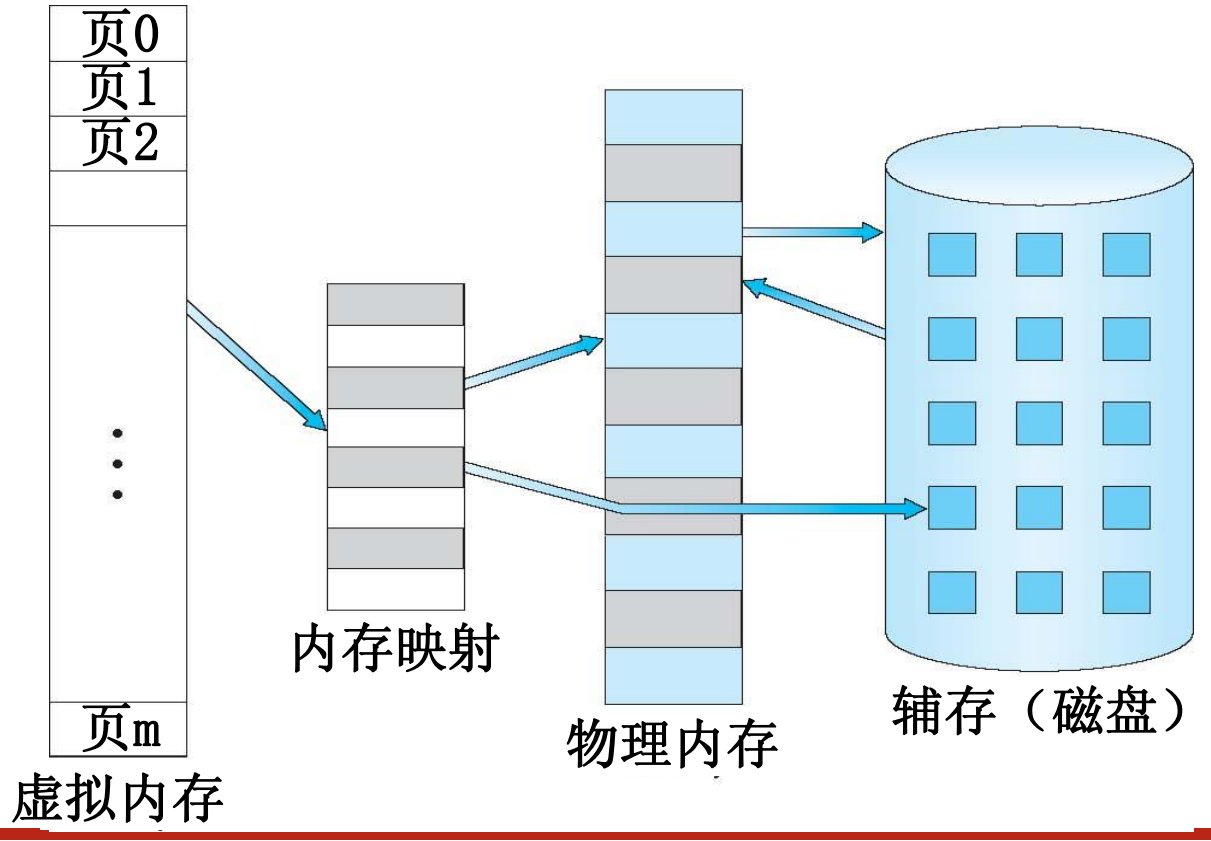
思考两个小问题

■ 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

cs:ip

逻辑地址

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

物理地址

0x7008

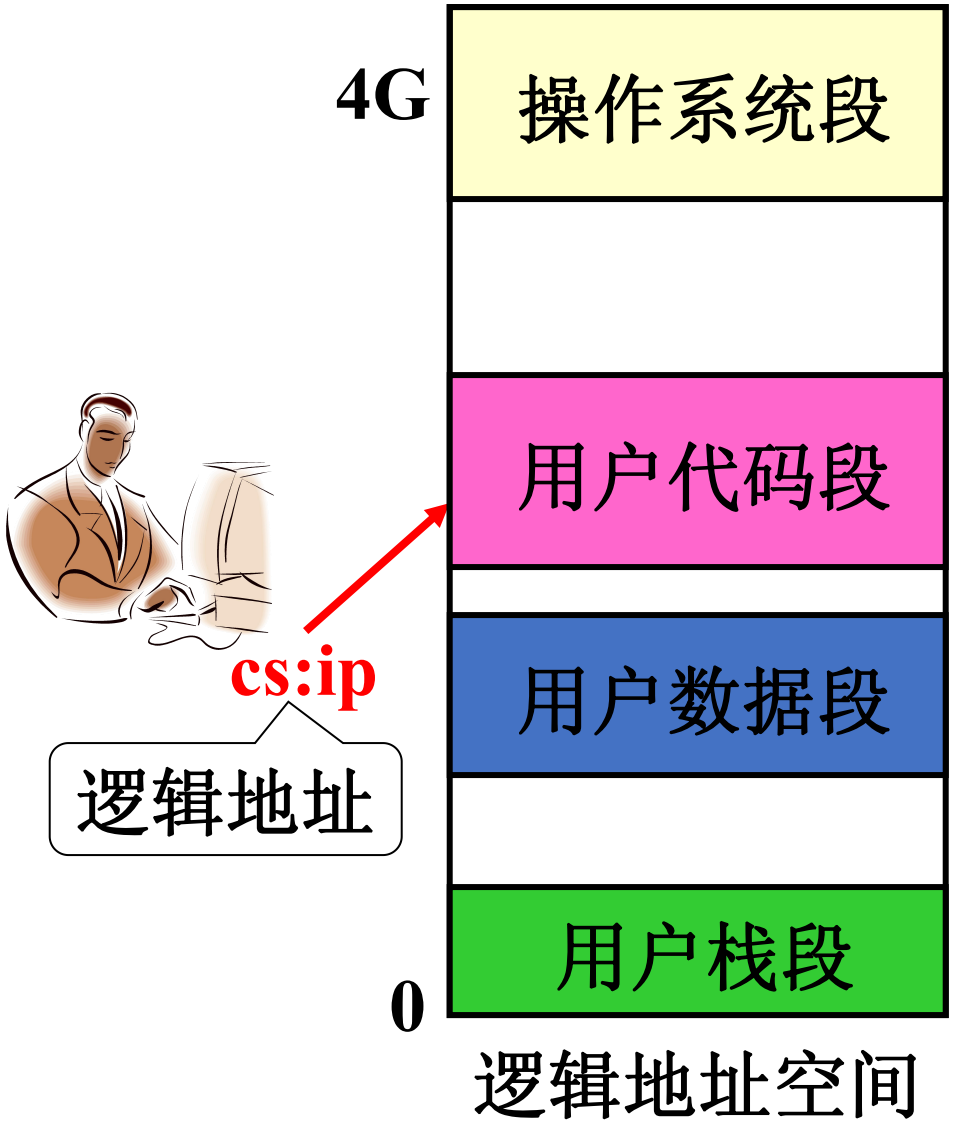
0x00345008

线性地址

物理内存

对用户是透明的

用户眼里的内存!



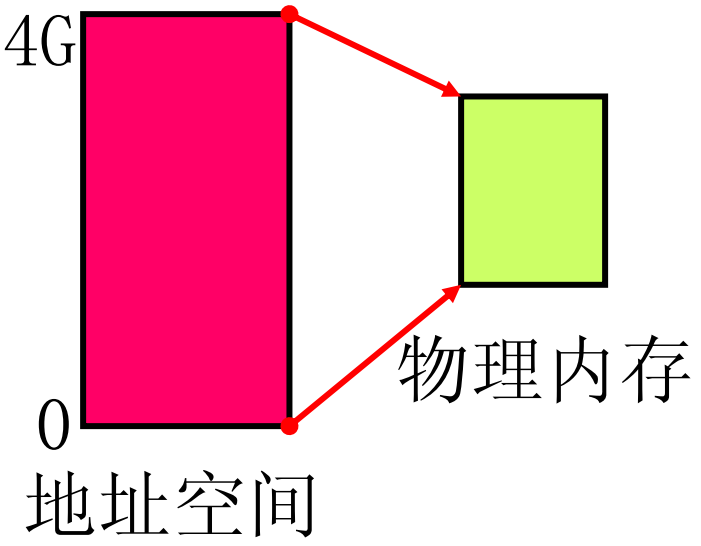
- 1个4GB（很大）的地址空间
- 用户可随意使用该地址空间，就象单独拥有4G内存
- 这个地址空间怎么映射到物理内存，用户全然不知

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

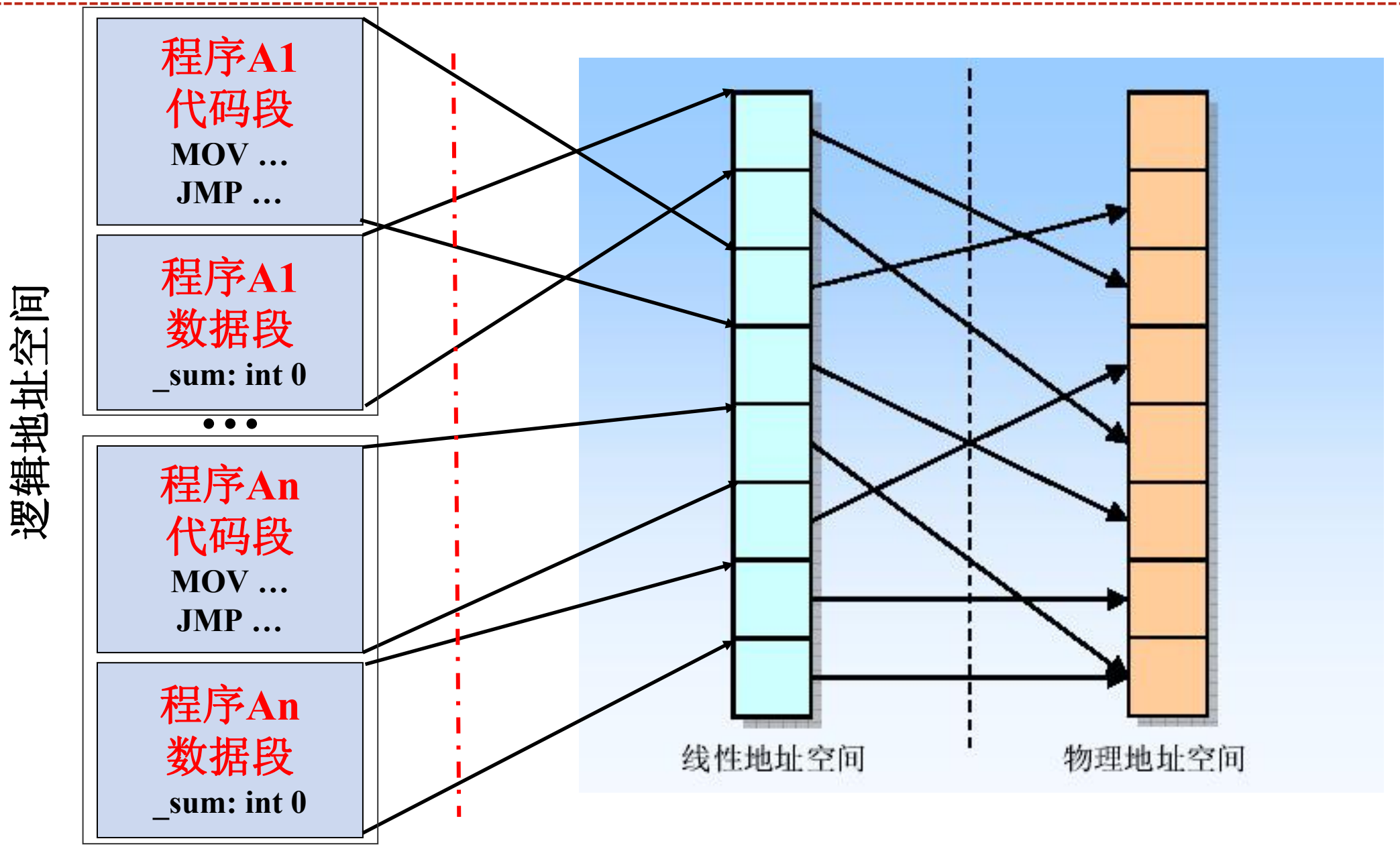
- 优点1: 地址空间>物理内存
  - 用户可以编写比内存大的程序
  - 4G空间可以使用，简化编程
- 优点2: 部分程序或程序的部分放入物理内存
  - 内存中可以放更多进程，并发度好，效率高
  - 将需要的部分放入内存，有些用不到的部分从来不放入内存，内存利用率高
  - 程序开始执行、响应时间等更快



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


逻辑地址空间
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
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。

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

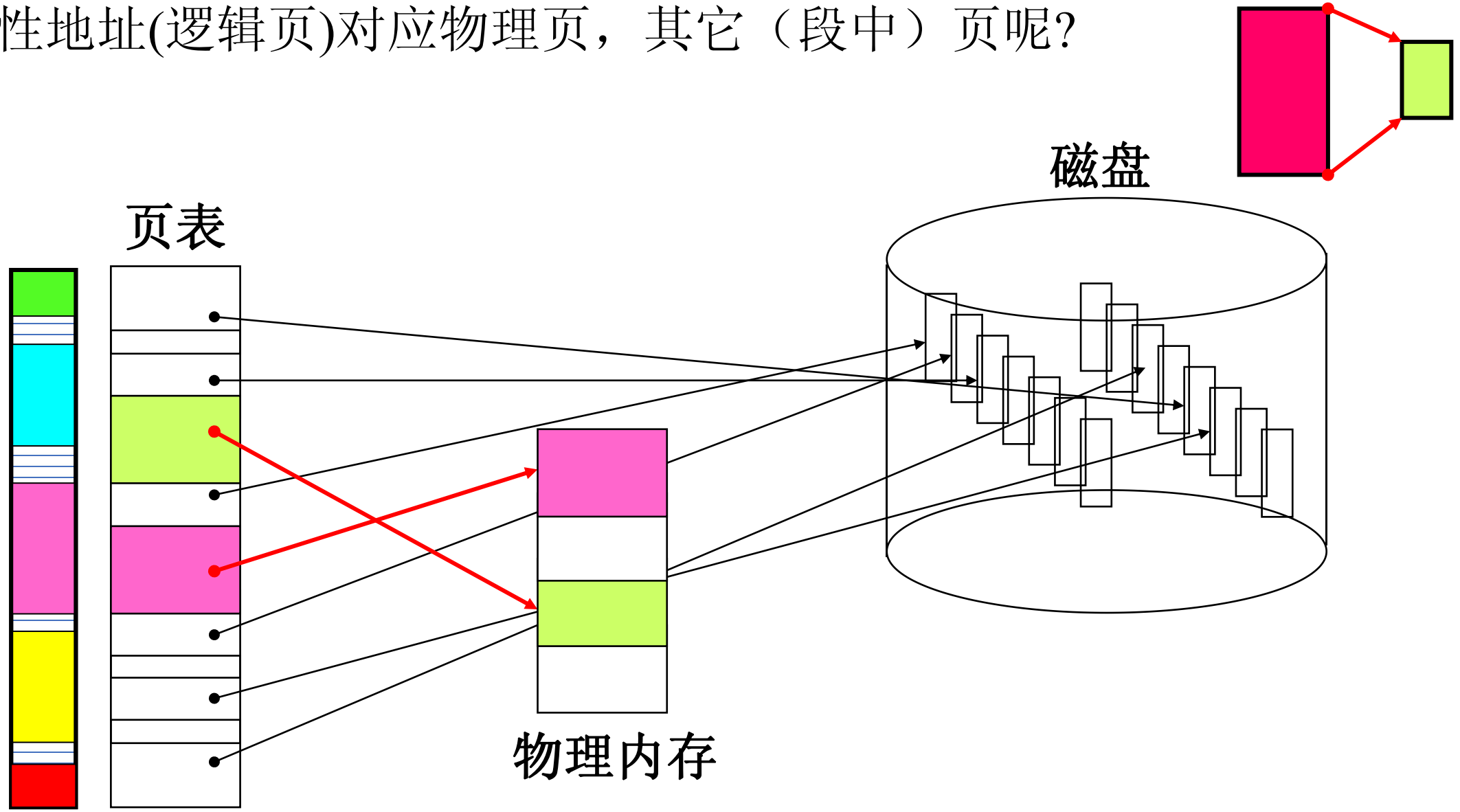
物理页号	偏移

物理地址

分页易于硬件实现、对用户透明，适合请求调入

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?

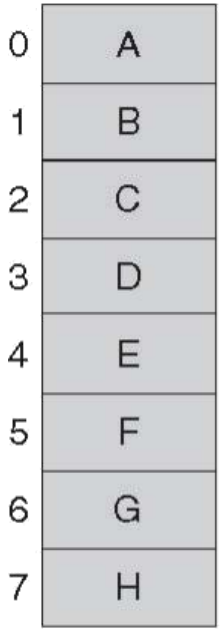


如何记录页是否在内存?

帧号 有效/无效位

	v
	v
	v
	v
	i
……	
	i
	i

改造后的页表



逻辑内存

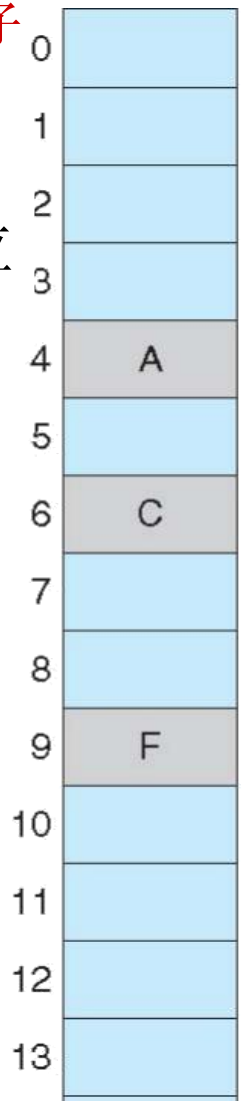
某些页不在内存中时的页表

有效-无效位

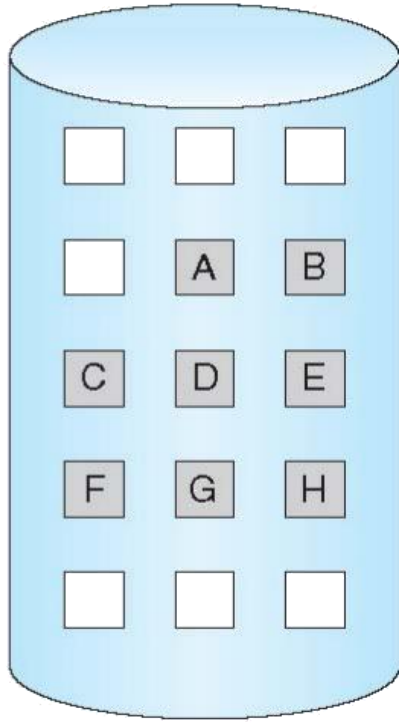
帧号

0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

页表



物理内存

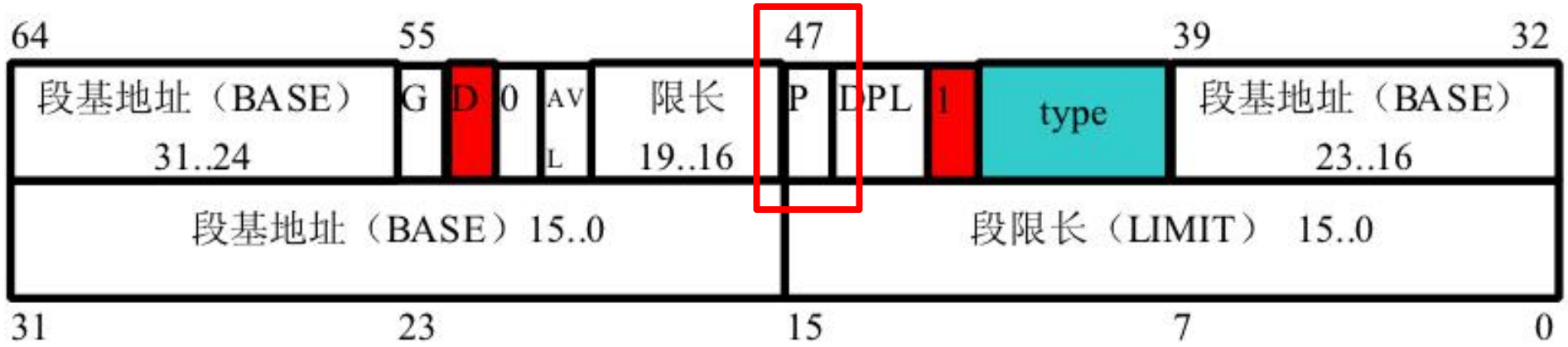


辅存（磁盘）

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

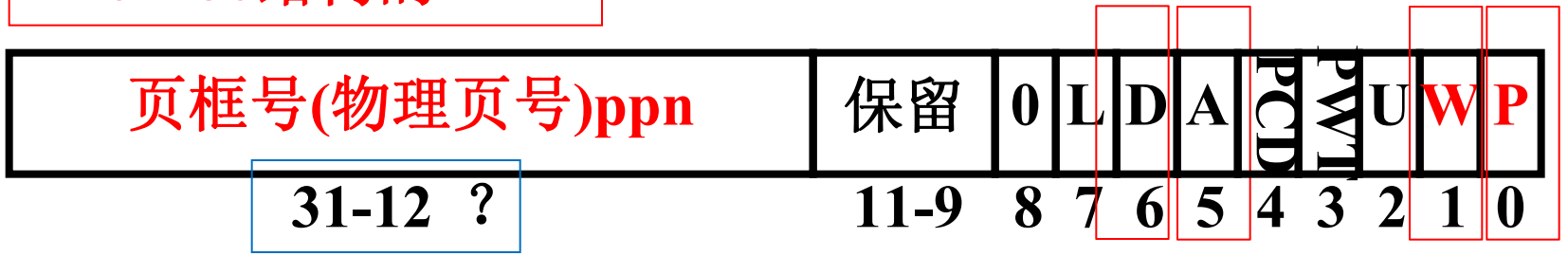
32位保护模式下的寻址方式

	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。

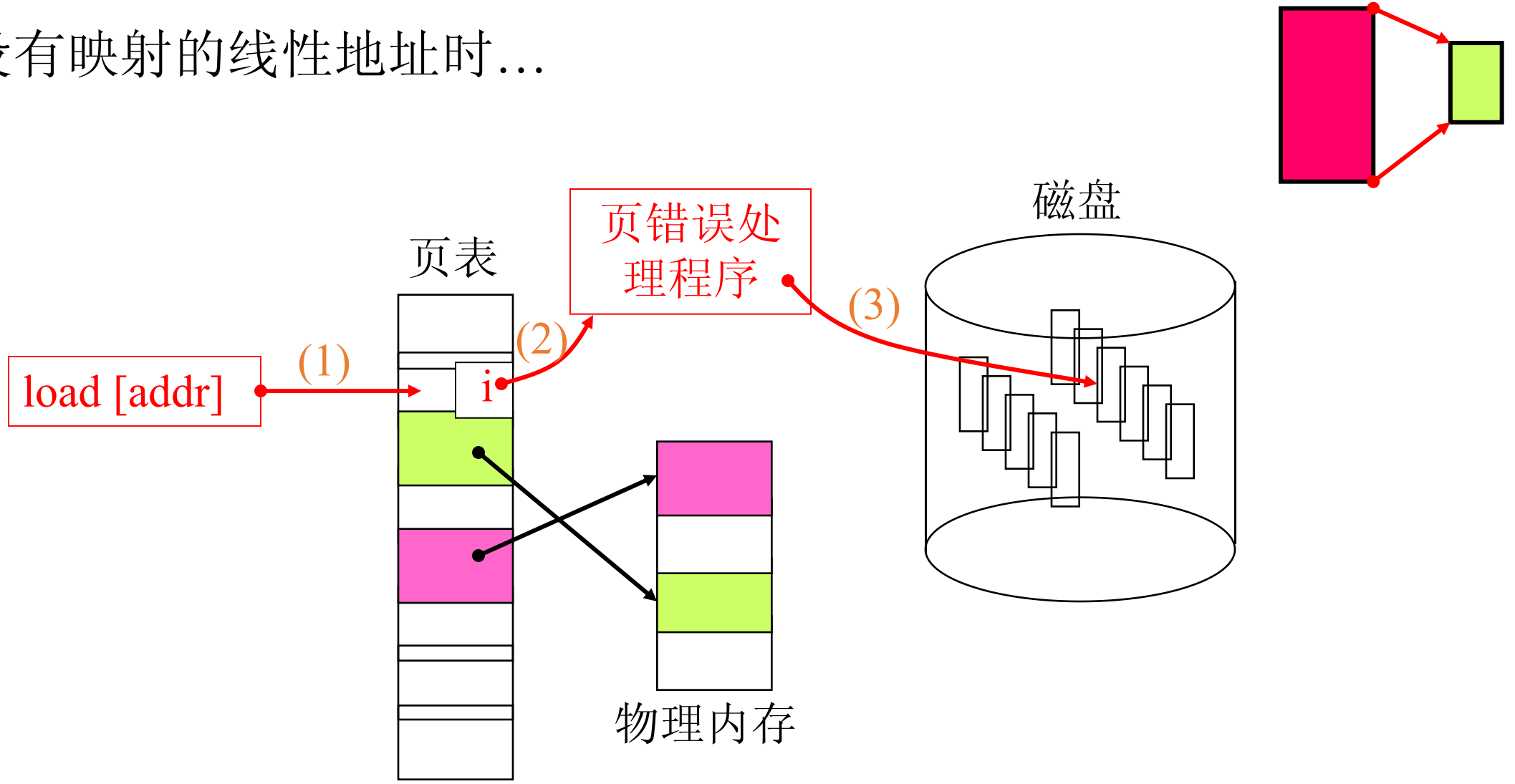
D--位6是页面已被修改（Dirty）标志。

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

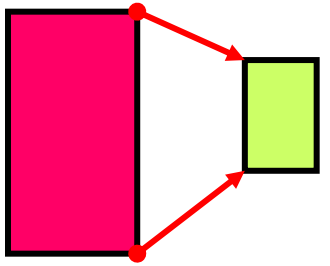
请求调页过程

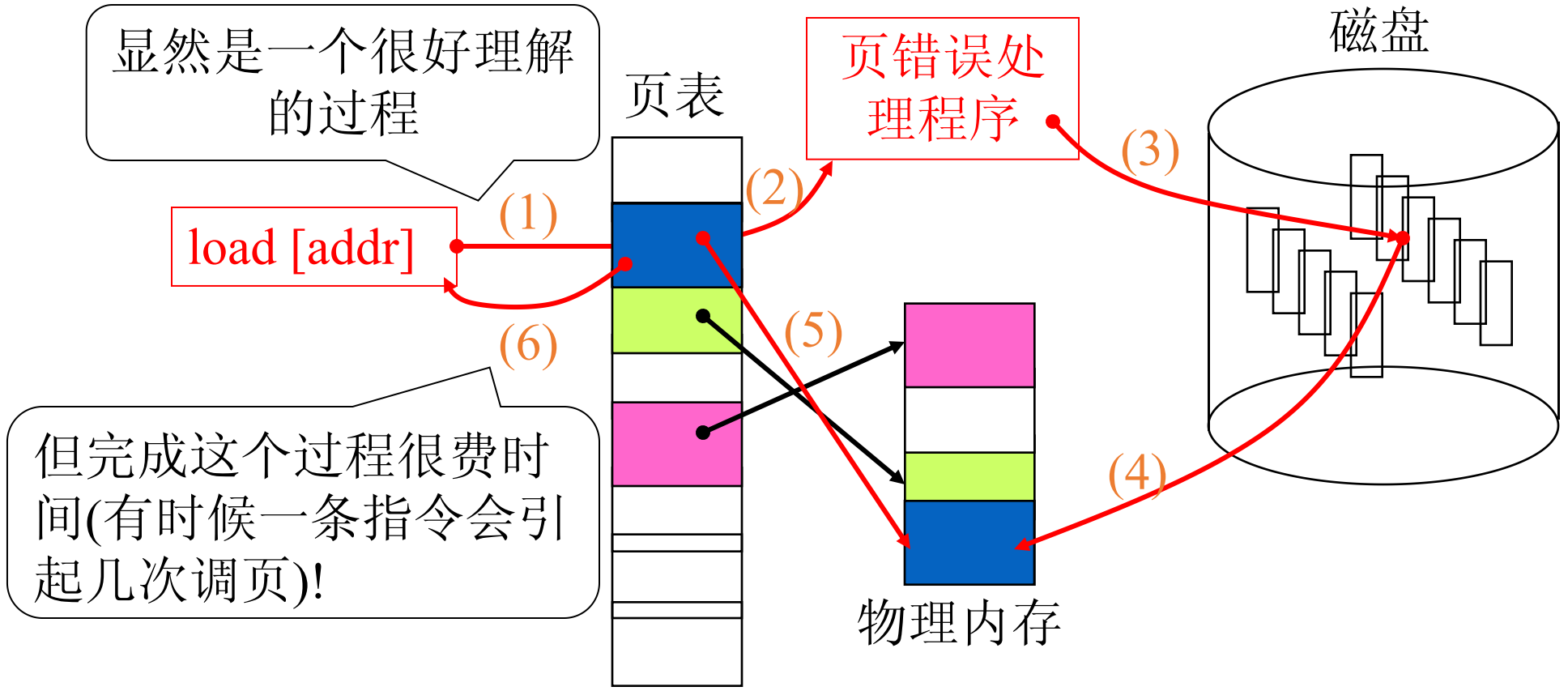
- 当访问没有映射的线性地址时…



请求调页过程

- 当访问没有映射的线性地址时…





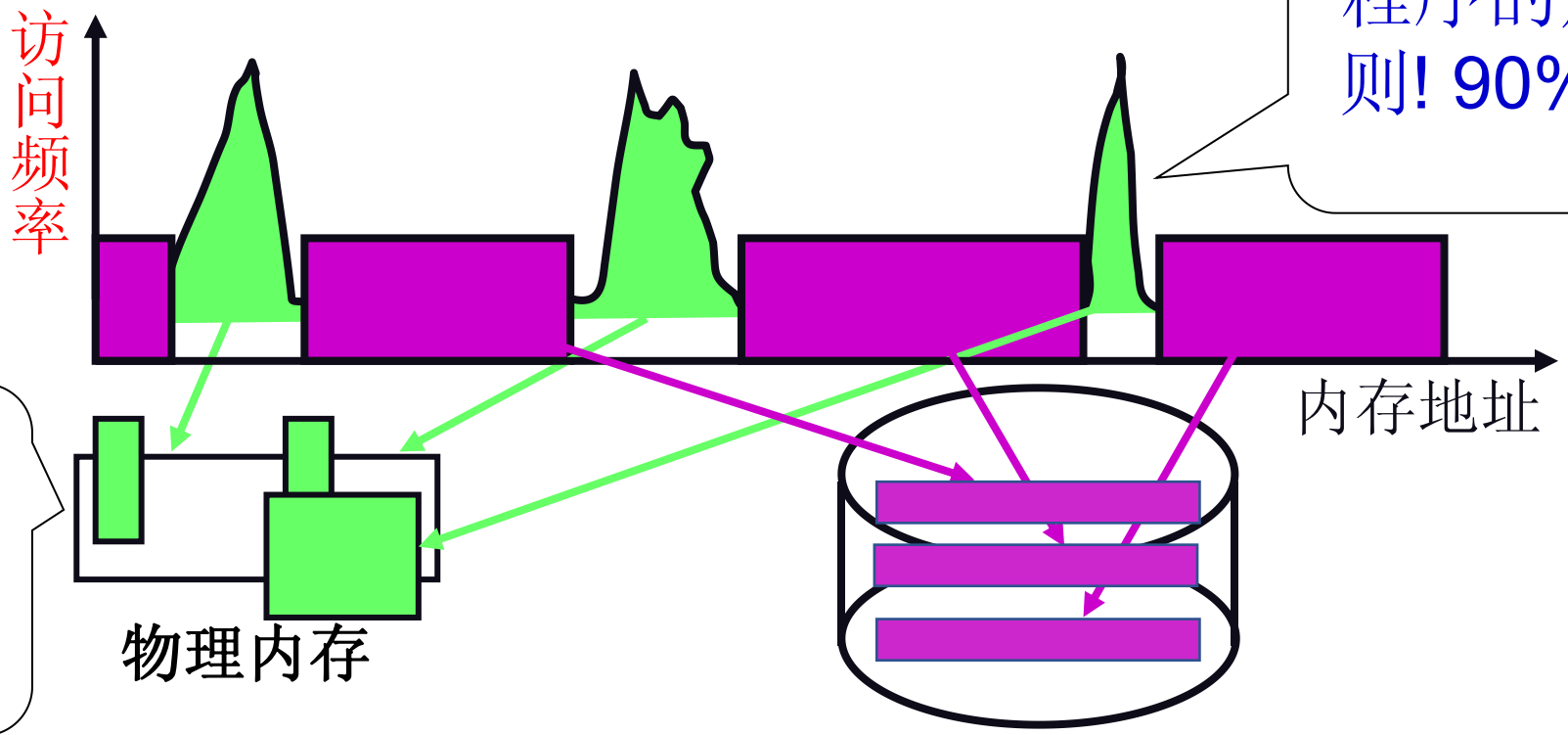
请求调页为什么可行?

- 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!

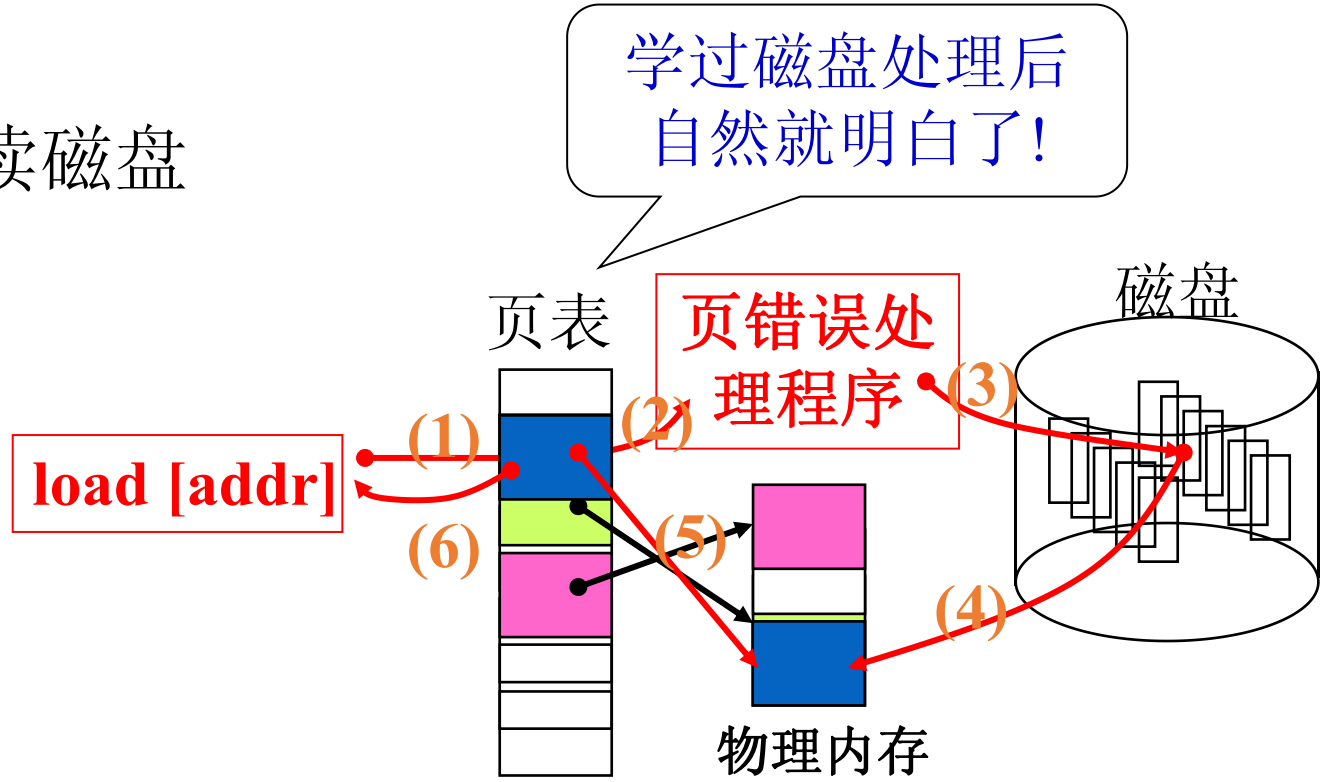


页面4K，调入一页后许多指令不出现页错误

这些参数从实践中获得!

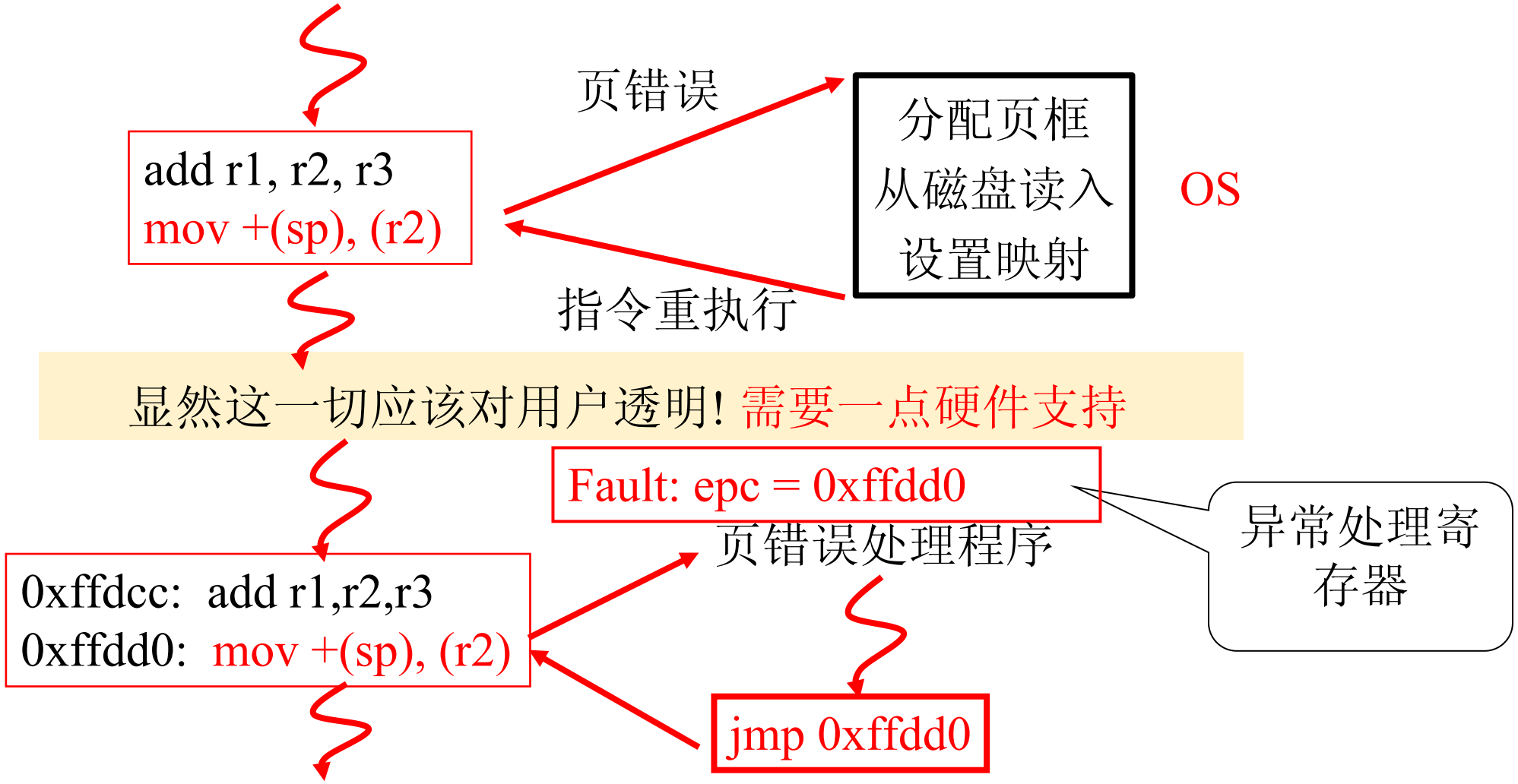
请求调页的具体实现细节

- (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



如何重新开始指令?

- 在指令执行过程中出现页错误



如何选一个空闲页框?

- 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（**置换**）
  - 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
- 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…

虚拟内存置换算法

- （1）先进先出页面置换 First In First Out (FIFO)
- （2）最优页面置换 Optimal replacement (OPT)
- （3）最近最少使用页面置换 （准确实现）Least-Recently-Used (LRU)
  - 计数器法
  - 页码栈法
- （4）最近最少使用页面置换（近似实现）Least-Recently-Used (LRU)
  - 引用位法
  - 时钟法

先进先出页面置换 FIFO

- 先进先出算法淘汰最早调入的页面

  一实例: 分配了3个页框(frame)，需要调用4个页面，它们的引用序列为

  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置换

最佳页面置换 OPT

- 最佳页面置换算法: 选未来最远将使用的页淘汰。

  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					

最佳页面置换算法是Belady于1966年提出的一种理论上的算法。是一种保证最少的缺页率的理想化算法。

- 本实例，OPT导致5次缺页 < FIFO (7 次)
- 是一种最优的方案，可以证明缺页数最小!
- 可惜，OPT需要知道将来发生的事… 怎么办?

用过去的历史预测将来。

最近最少使用页面置换 (LRU)

- 最近最少使用算法：选最近最长时间没有使用的页淘汰（ OPT的可实现的近似算法）。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 = OPT (5次)
- LRU是公认的很好的页置换算法，怎么实现？

计数器，页码栈

LRU的准确实现方法1：计数器法

- 每页维护一个时间戳（time stamp），即计数器
- 具体方法：设一个时钟(计数)寄存器，每次页引用时，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

计数器（每次引用，加1）

	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

- 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

- 维护一个页码栈
  - 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

怎么办呢？

- 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

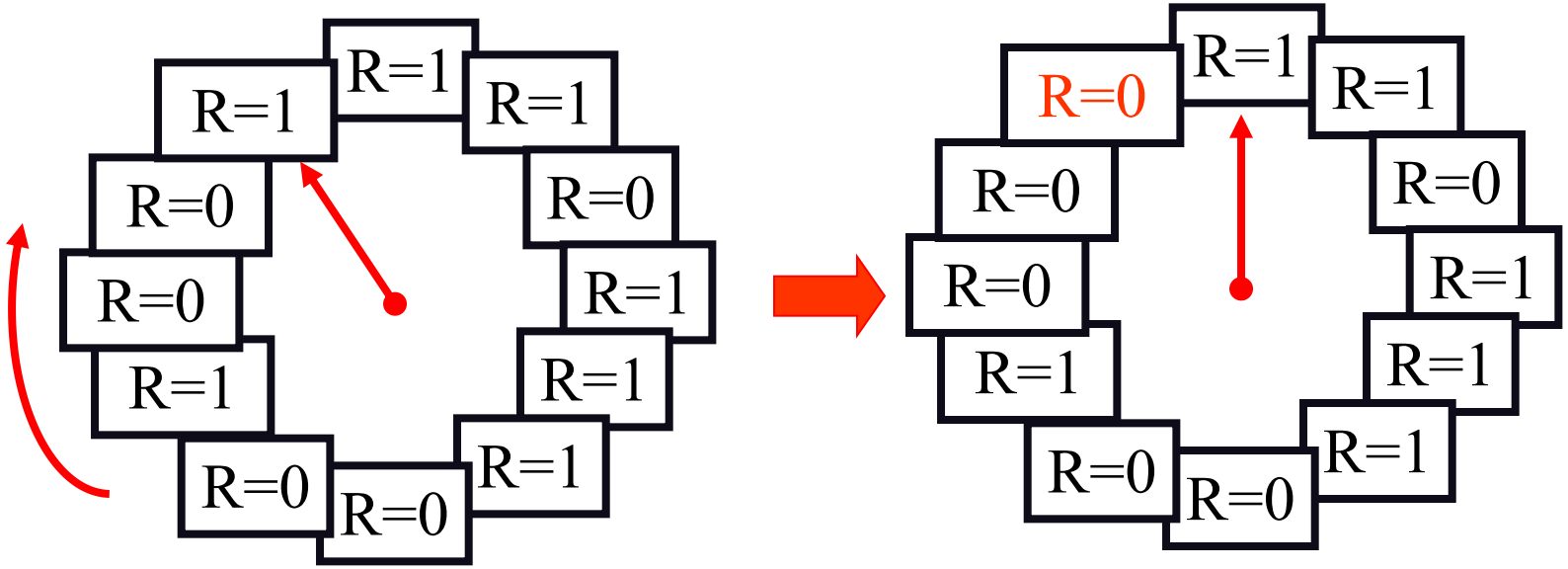
退而求其次：降低准确率，提高效率⇒LRU近似实现

LRU近似实现：再给一次机会算法

- 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实现方法称为时钟算法Clock Algorithm

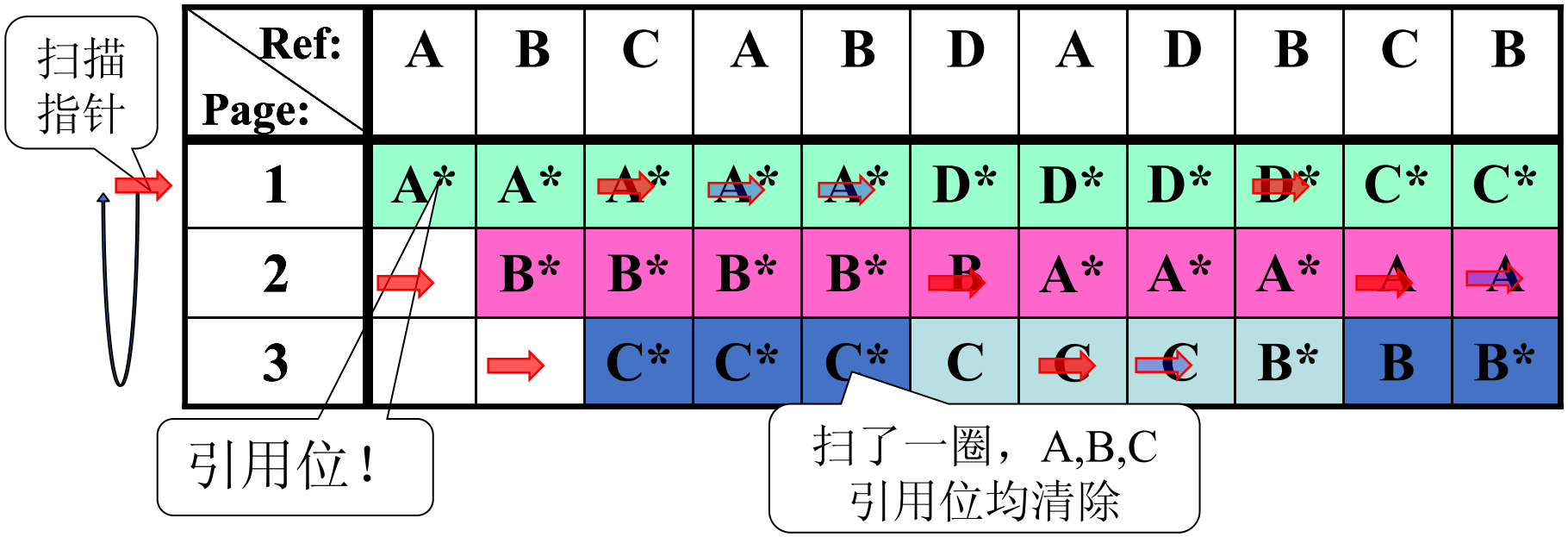
Clock算法实例

- 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一页。



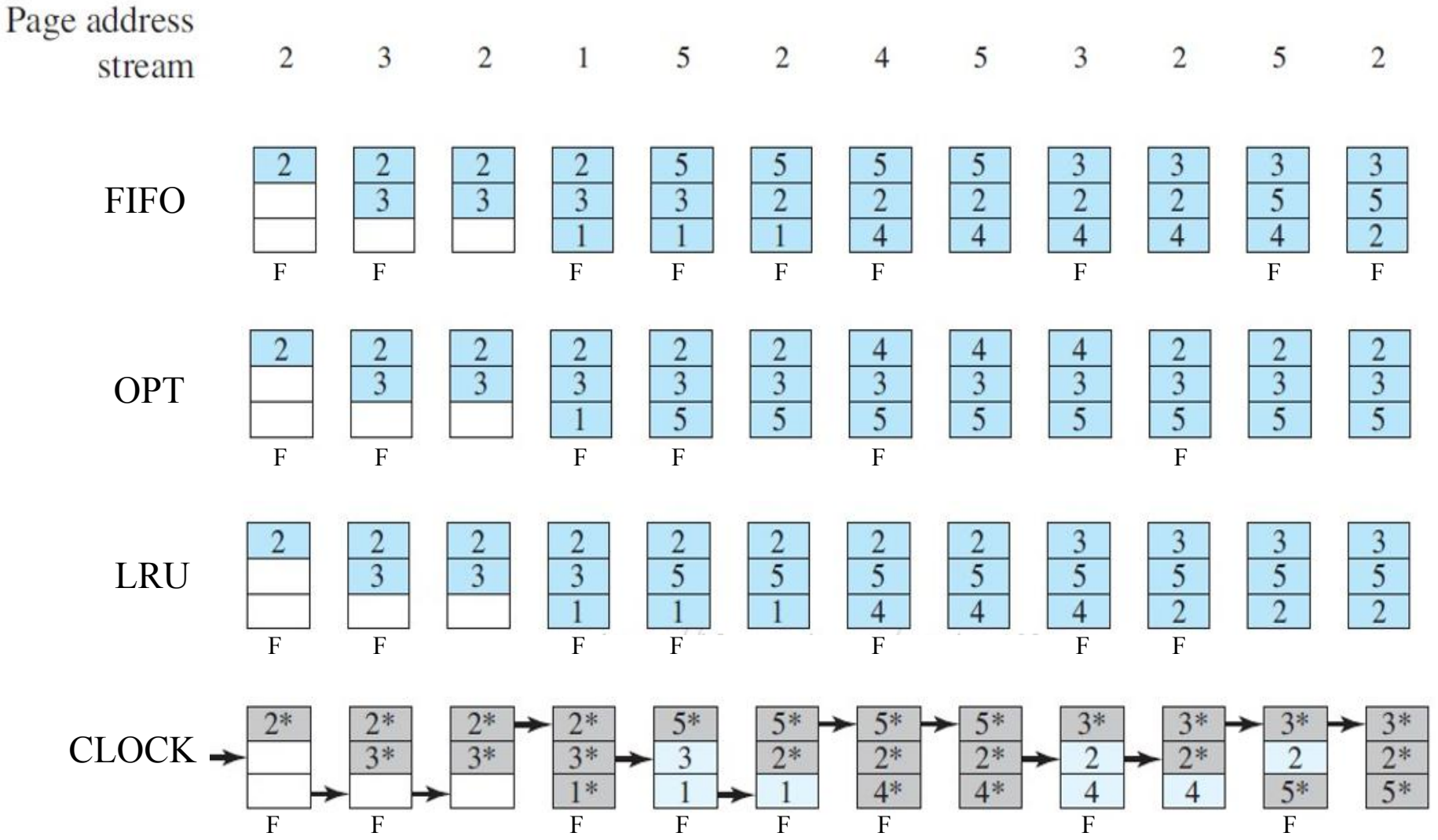
所有的R=1，原因: 记录了太长的历史信息，指针扫描一圈后淘汰当前页，指针前移一位。退化为FIFO!

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

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:如存在页命中， 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）

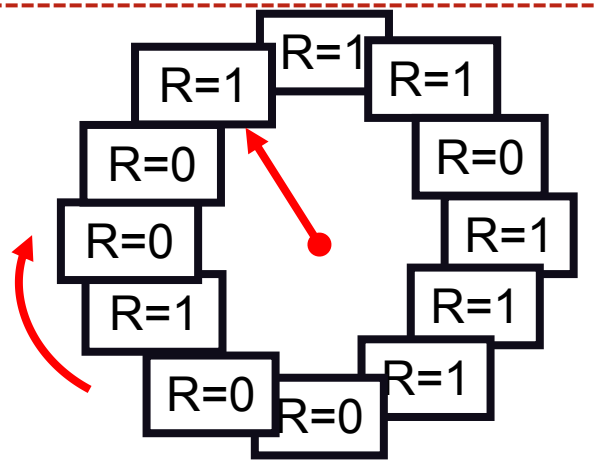
随堂习题

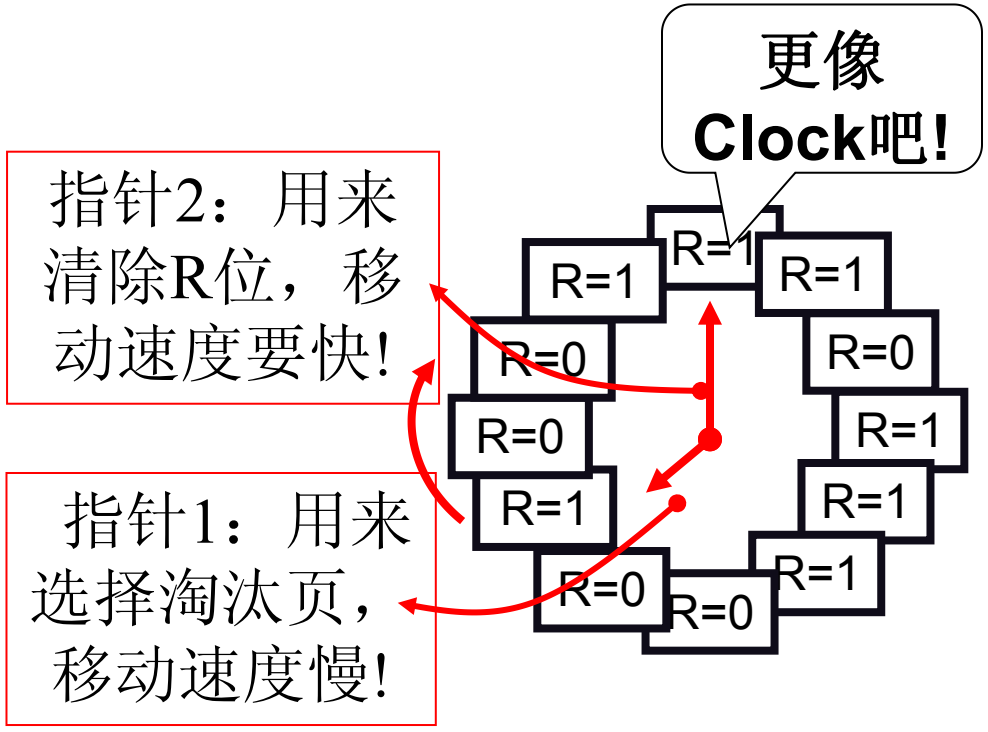

Page address stream
2 3 2 1 5 2 4 5 3 2 5 2
FIFO
2 2 2 2 5 5 5 5 3 3 3 3
3 3 3 3 2 2 2 2 2 4 5 5
1 1 1 4 4 4 4 5 2
F F F F F F F F
OPT
2 2 2 2 2 2 4 4 4 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3
1 5 5 5 5 5 5 5 5 5
F F F F F F
LRU
2 2 2 2 2 2 2 2 3 3 3 3
3 3 3 3 5 5 5 5 5 5 5 5
1 1 1 4 4 4 4 2 2 2
F F F F F F F F
CLOCK
2* 2* 2* 2* 5* 5* 5* 5* 3* 3* 3* 3*
3* 3* 3* 1* 3 2* 2* 2* 2 2* 2* 2*
1 1 4* 4* 4 4 5* 5*
F F F F F F F F

Clock算法分析的改造

- 如果缺页很少，会导致所有的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

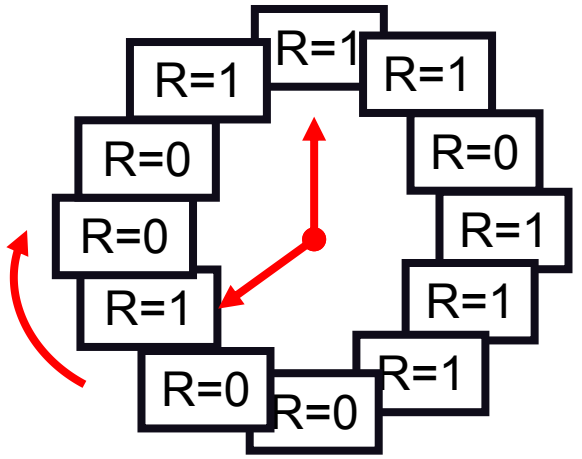
清除R位的hand如何定速度，若太快?
又成了FIFO



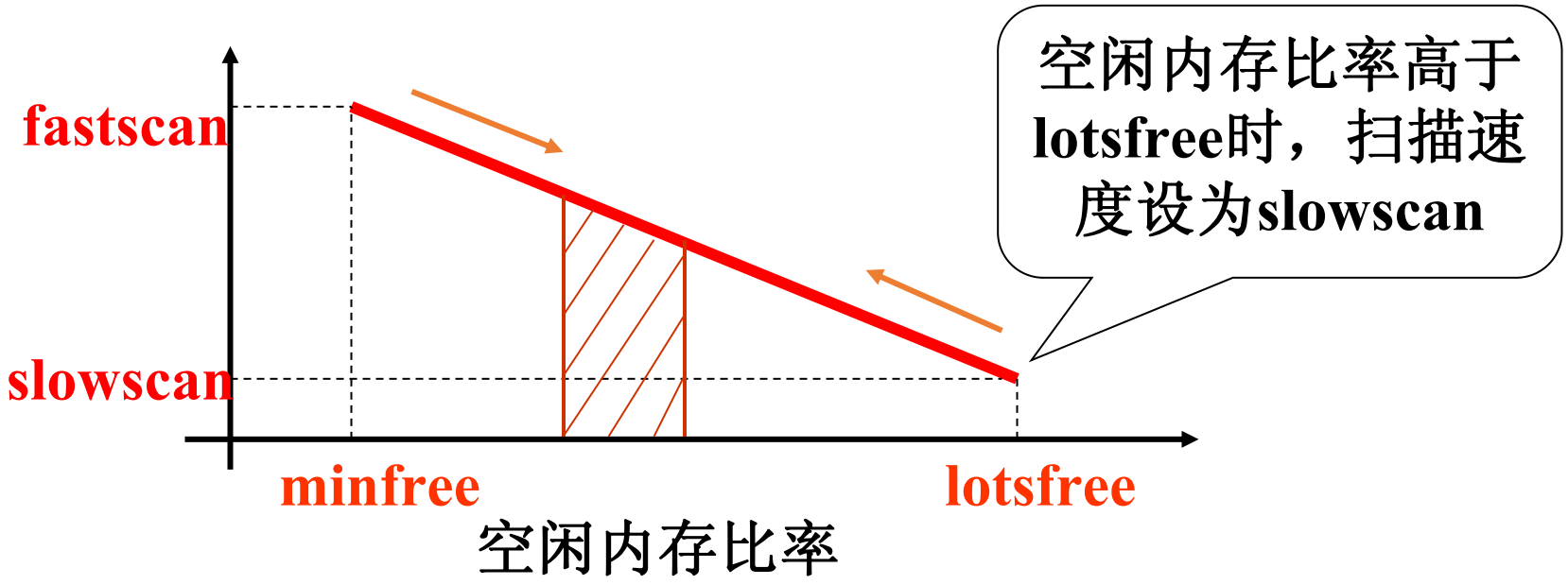


Solaris实际做法

- 清除R位的hand速度设固定值吗?
- 系统负载并不固定…



在slowscan和fastscan之间调整

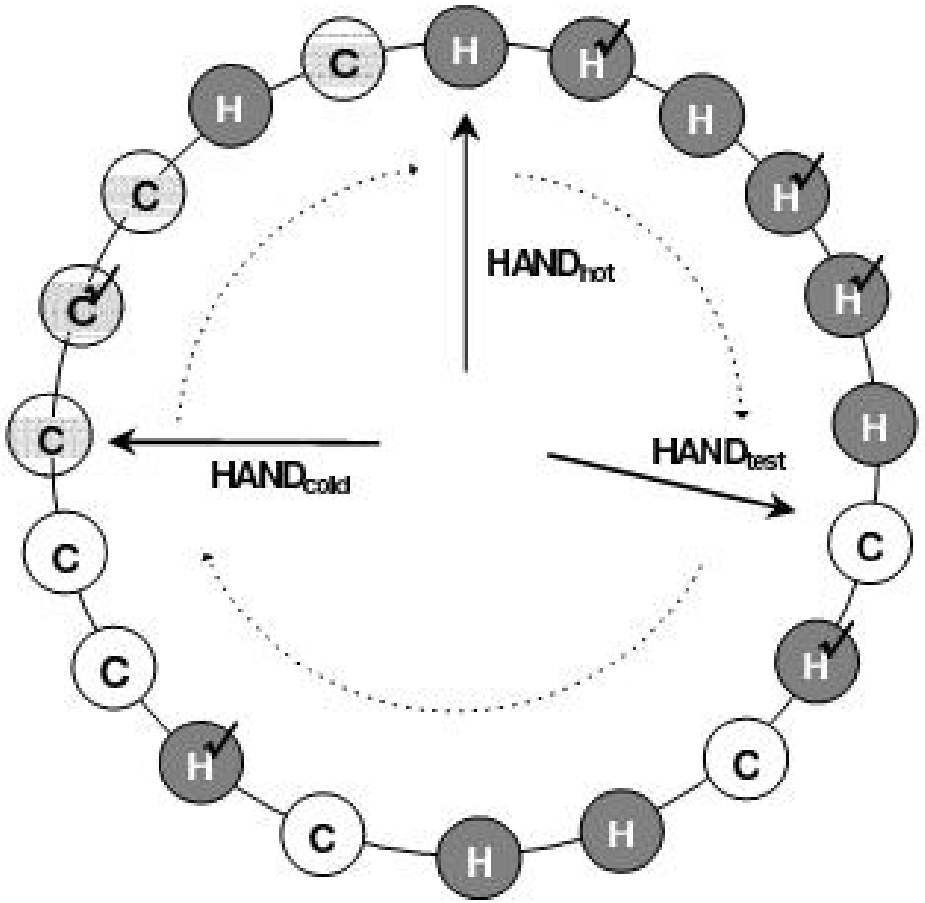


改进型Clock置换算法

- 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)最近已被访问且已被修改
- 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）

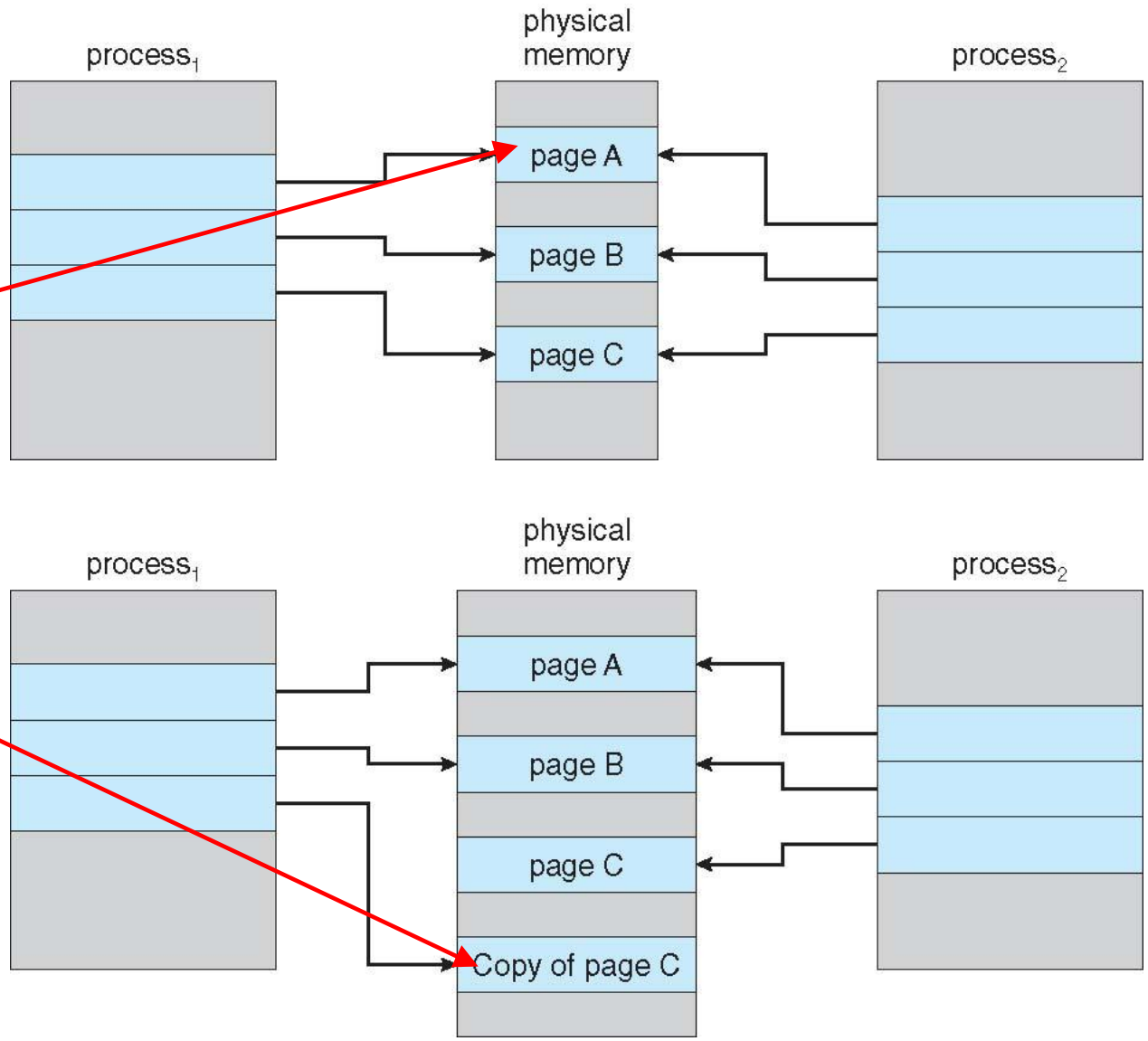
- Efficient SSD Caching by Avoiding Unnecessary Writes using Machine Learning @ICPP 2018
- LIPA: A Learning-based Indexing and Prefetching Approach for Data Deduplication@MSST 2019
- A Survey of Machine Learning Applied to Computer Architecture Design @ https://arxiv.org/

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

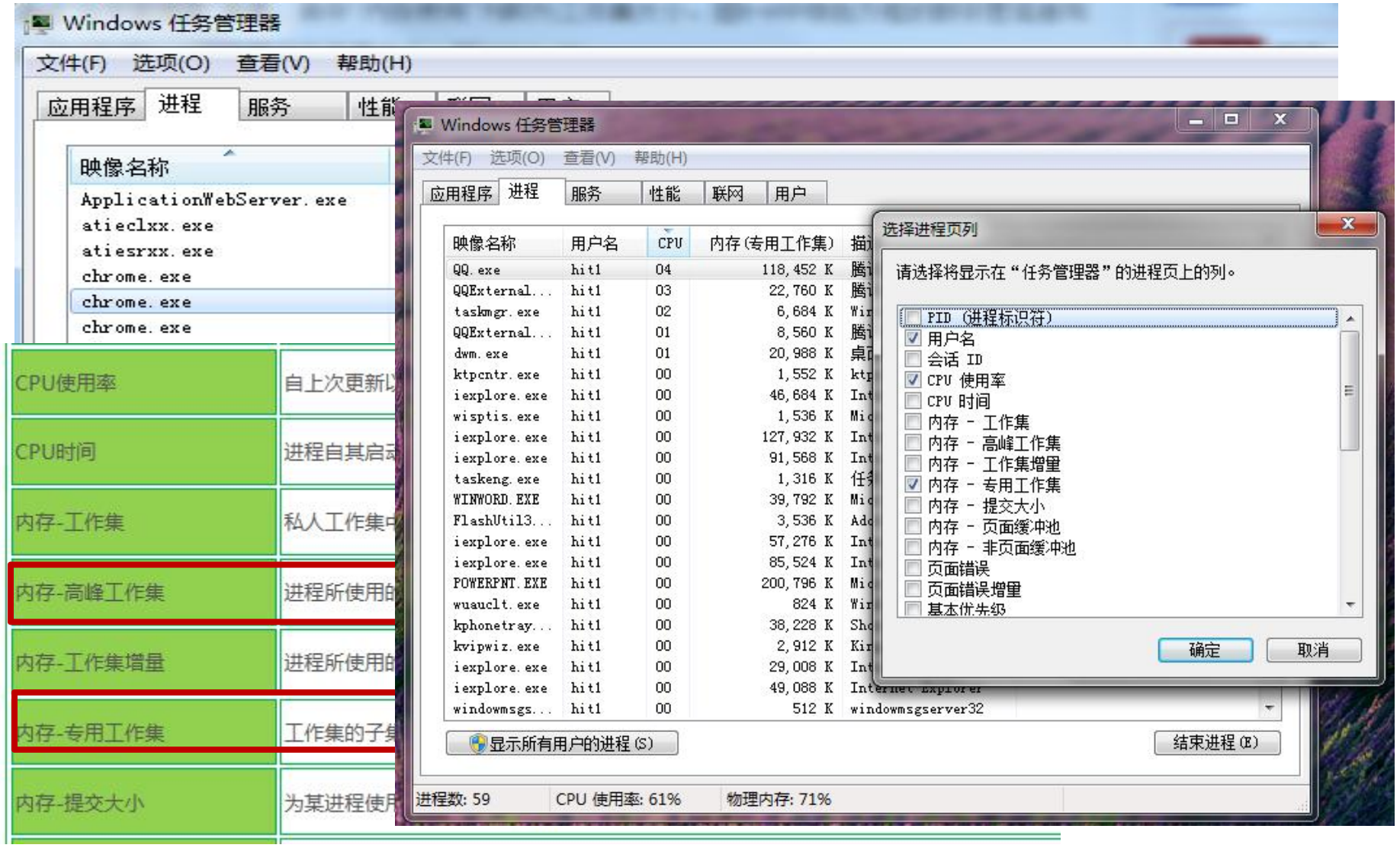
工作集

- 工作集（驻留集）：给进程分配的主存物理空间。是动态变化的。
- 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：
  - 1. 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。
  - 2. 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。
  - 3. 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

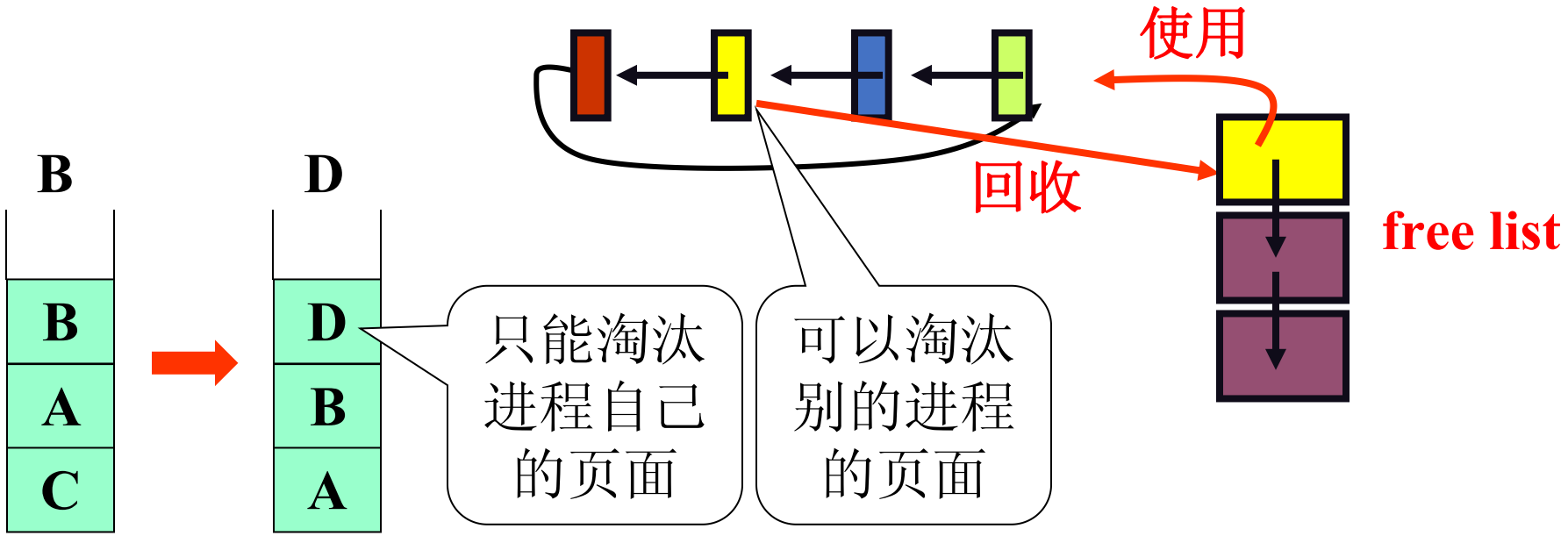
- Ps命令是进程查看命令
- RSS进程使用的驻留集大小或者是实际内存的大小，Kbytes字节。

```
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]
```

- RSS 是常驻内存集（Resident Set Size），表示该进程分配的内存大小；
- VSZ 表示进程分配的虚拟内存；

两种置换策略

- 全局置换
- 局部置换



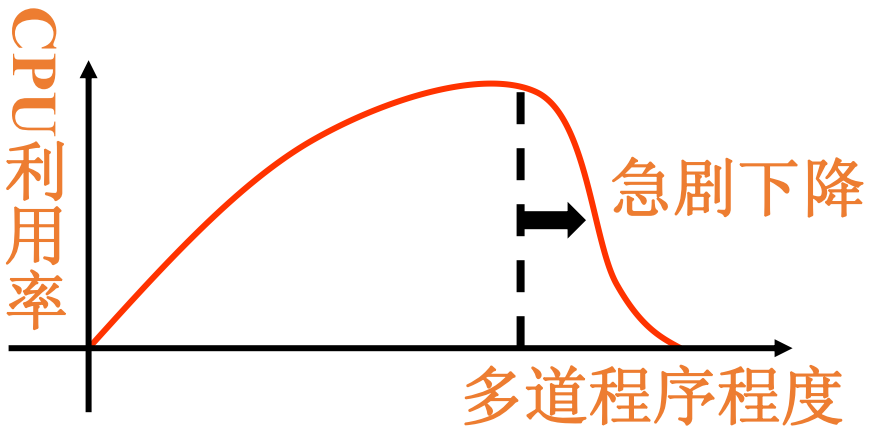
- 全局置换: 实现简单
- 但全局置换不能实现公平、保护： 一个经过巧妙优化的程序里会出现大量goto，则…

页面置换需要考虑的关键问题

- 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?

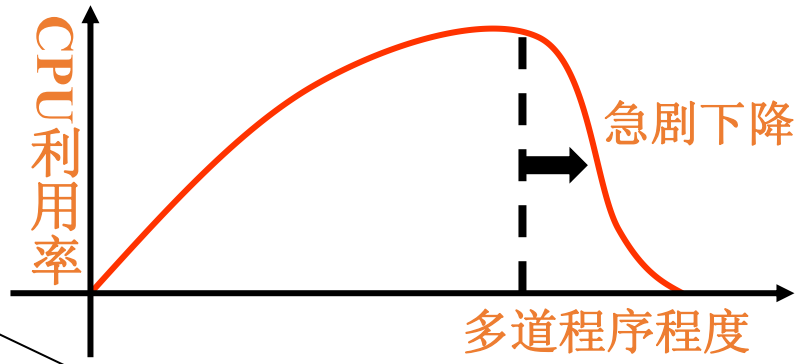
最坏情况需要6帧!

- 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

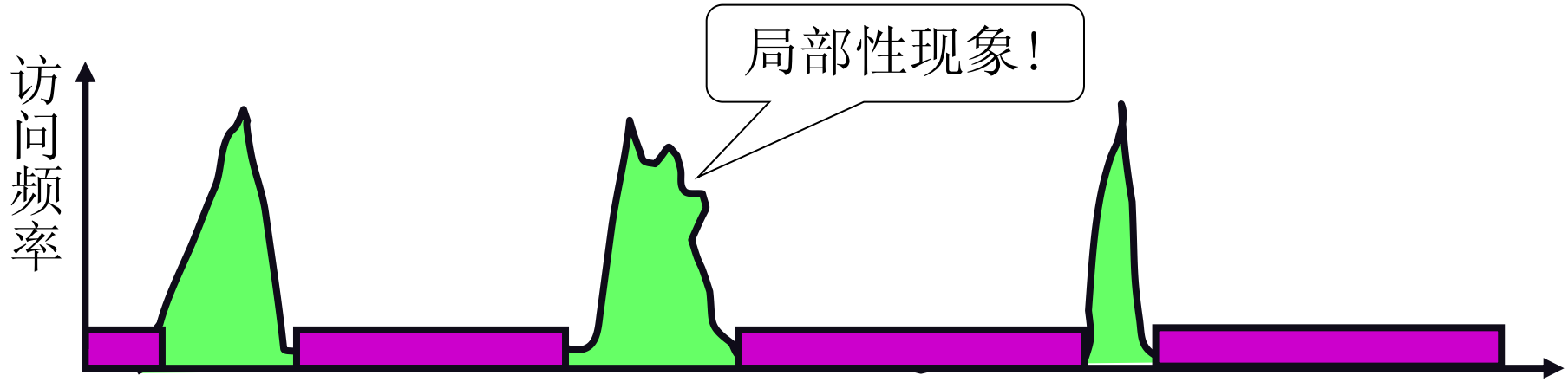


此时: 进程调入一页，需将一页淘汰出去，刚淘汰出去的页马上要需要调入，就这样……

问题是怎么确定进程需要多少帧才能不颠簸?

工作集模型

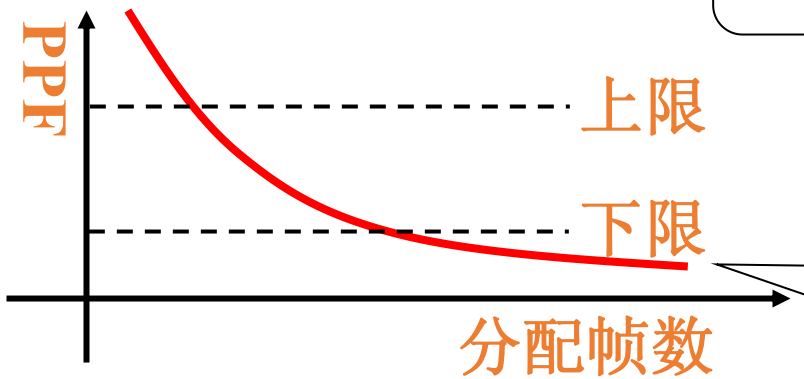
- 任何计算都需要一个模型! 要确定进程所需的帧数该依靠什么信息呢?
  - 从请求调页的可行性开始!



  - 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，PFF并不一定下降

- 此种方法简单直接，在处理颠簸时常用。
- 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

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

9faults

4frame

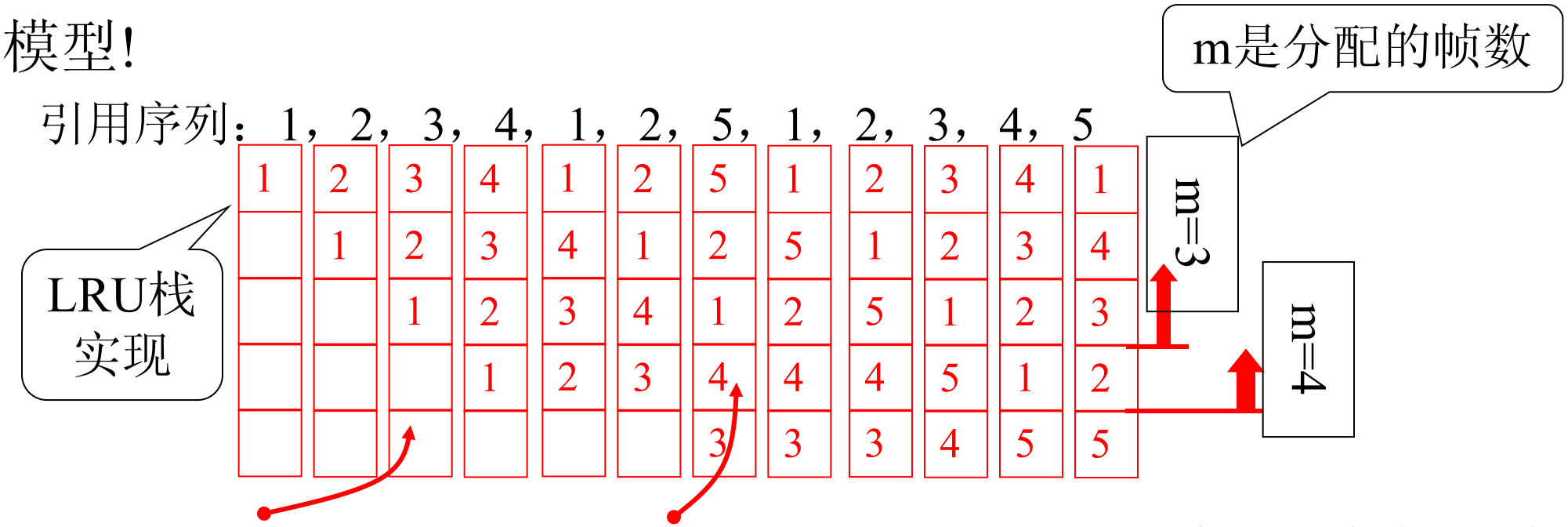
1	1	1	1	1	5	5	5	5	4	4	4
	2	2	2	2	2	1	1	1	1	5	5
		3	3	3	3	3	2	2	2	2	2
			4	4	4	4	4	3	3	3	3

10faults

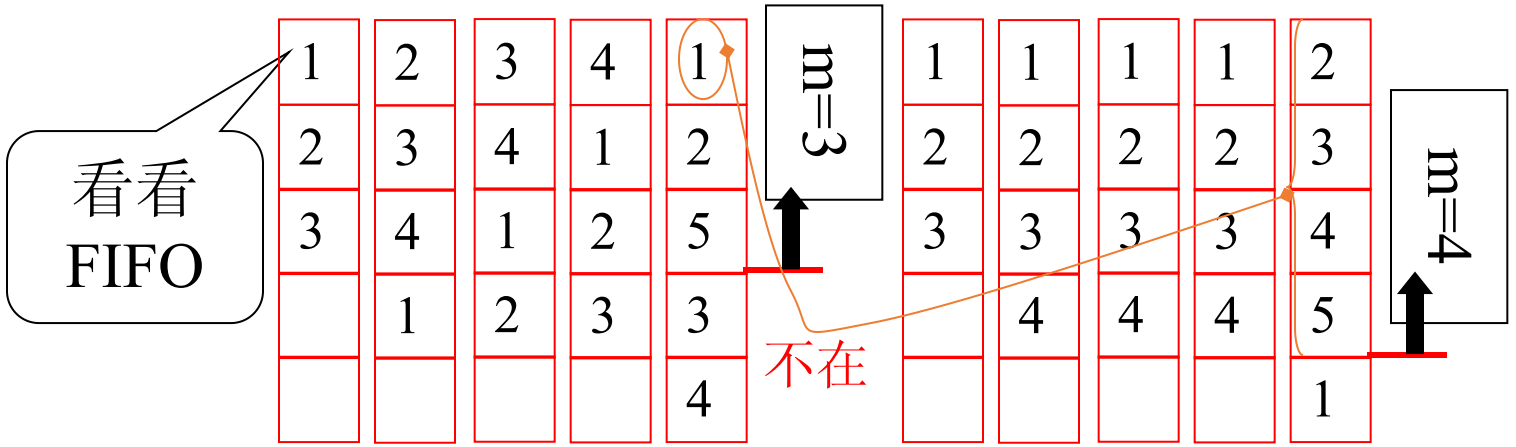
- Belady异常现象：对有的页面置换算法，页错误率可能会随着分配帧数增加而增加。

什么样的页置换没有Belady异常

■ 看个模型!



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!



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

- 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
- 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）

what?

内存利用率高，
程序编制容易，
响应时间快…

why?

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→OPT→LRU→Clock（NRU）
- 需要给进程分配页框 ⇒ 全局、局部 ⇒ 颠簸 ⇒ 工作集

课堂练习

1. 某虚拟存储器系统采用页式内存管理，使用FIFO页面替换算法，考虑页面访问地址序列3 8 1 7 8 2 7 2 1 8 3 1 2 1 3 1 7 1 2 8。假定内存容量为4个页面，开始时是空的，则页面失效次数是（ ）

A．5　　B．6　　C．7　　D．8

课堂练习

■ 2. 页面策略替换中可能引起CPU抖动（颠簸）的是（ ）

A. FIFO　　B. LRU　　C. 都不会　　D. 都会

Hope you enjoyed the OS course!