

# 第9章 虚拟内存

孙承杰

E-mail: [sunchengjie@hit.edu.cn](mailto:sunchengjie@hit.edu.cn)

哈工大计算学部人工智能教研室

2023年秋季学期

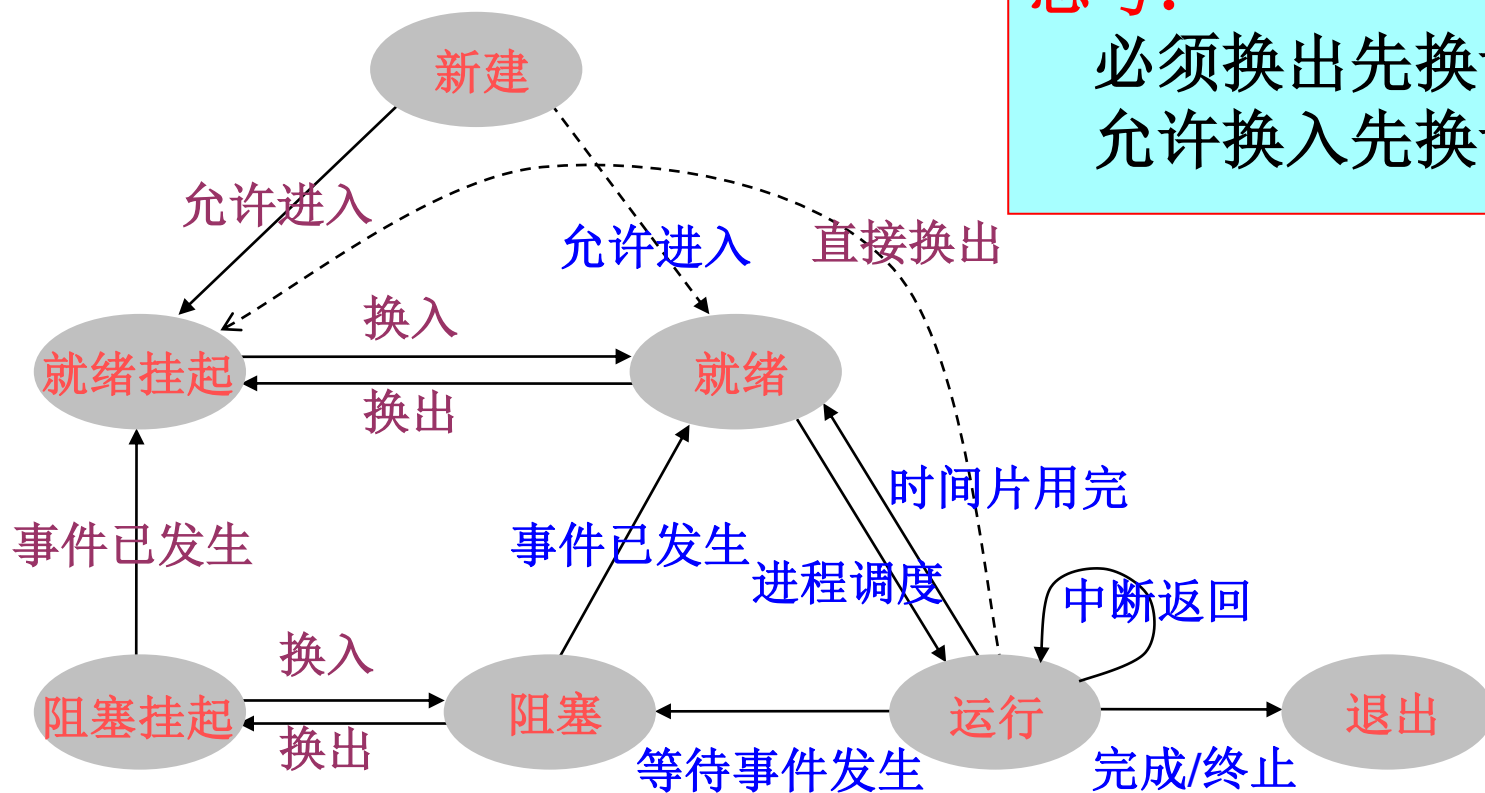


# 虚拟内存

程序部分加载 按需调页 换入换出

# 回顾：3.3 进程的描述与表达

## 进程七状态变迁图



进程状态变化图（七状态）

**挂起状态** - 引入主存 $\leftrightarrow$ 外存的交换机制，虚拟存储管理的基础

# 主要内容

## 9.1 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

## 9.2 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

## 9.3 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）

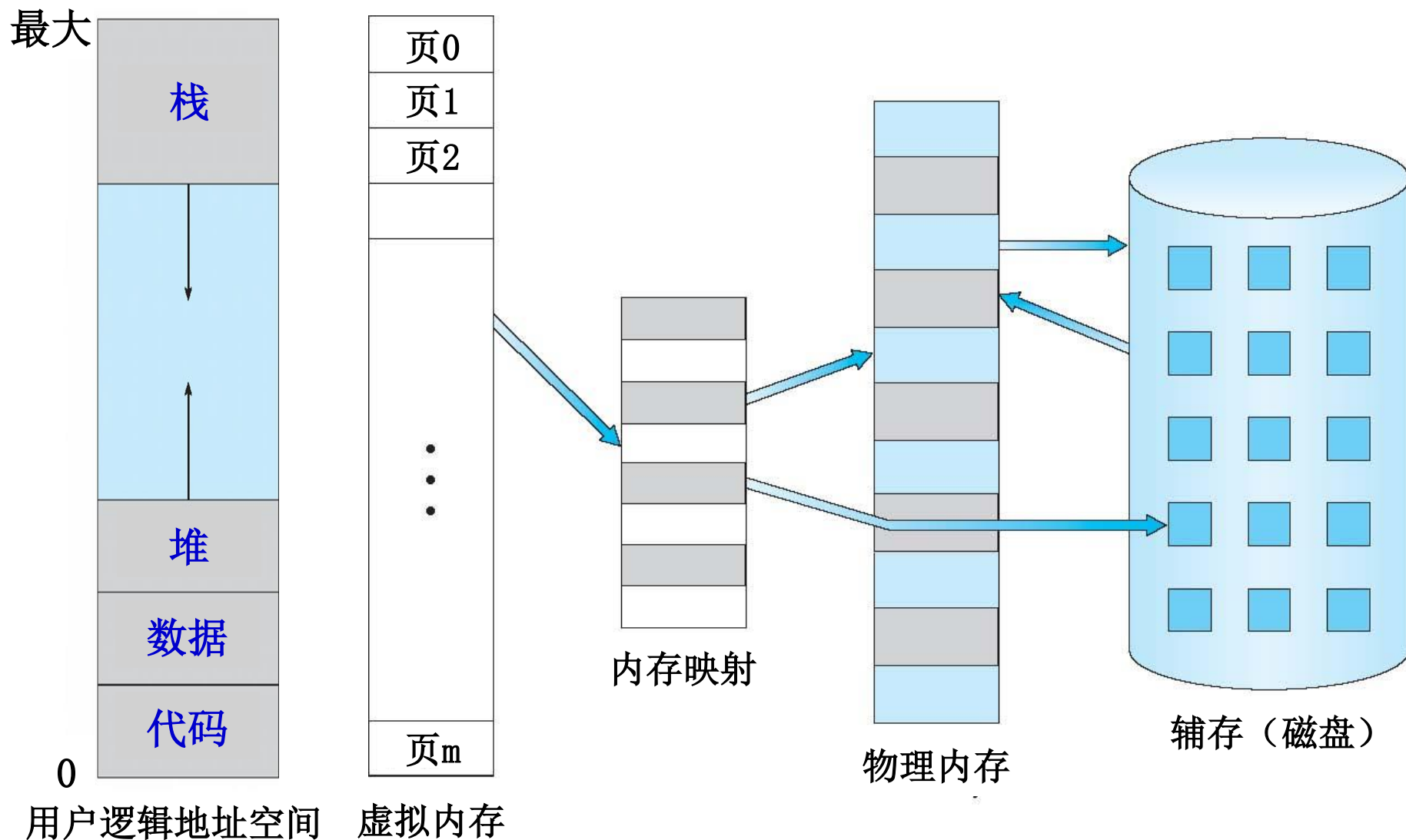
## 9.4 其他相关问题

- (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

## 9.1 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？多级缓存思想？



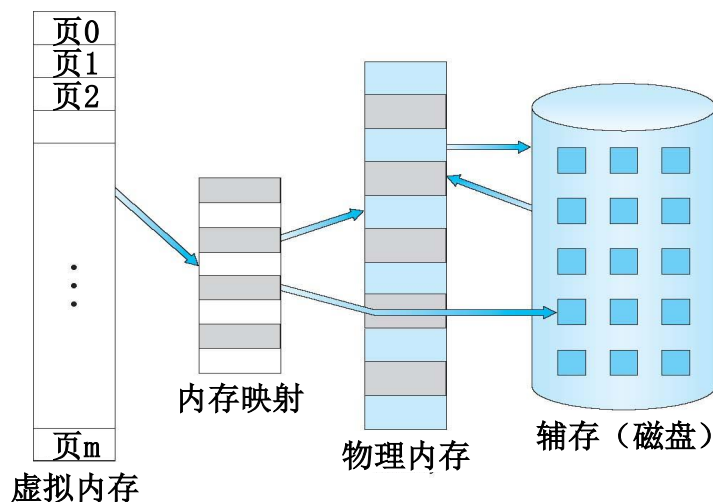
# 思考两个现象

一个实际的问题：

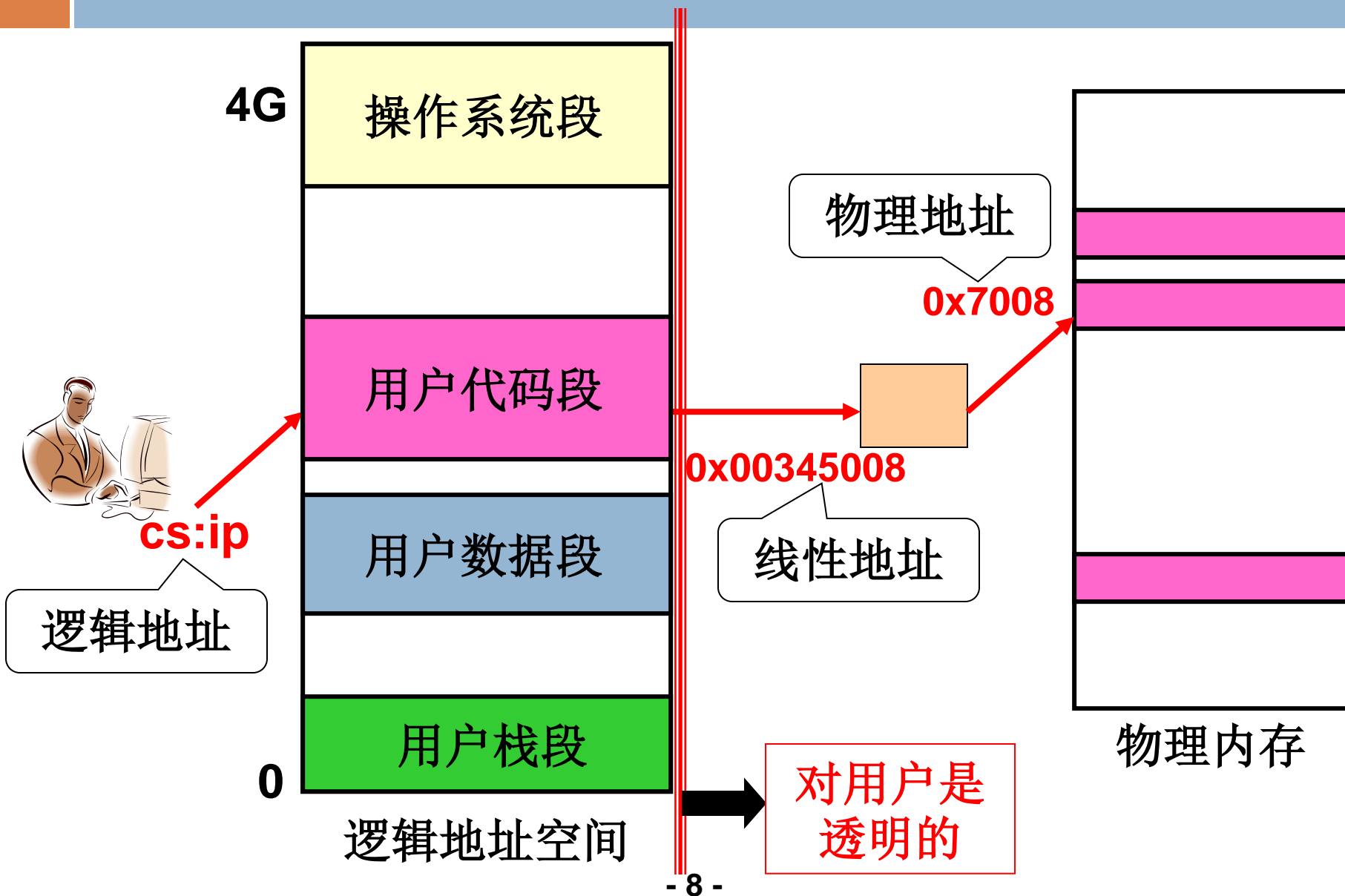
用**word**打开一个几百页的文档（至少几百兆）。在打开，前后拖拽；长时间不用或停留在某个页（操作），再拖拽。几种情况下，会发生什么现象，操作系统要做什么？

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？

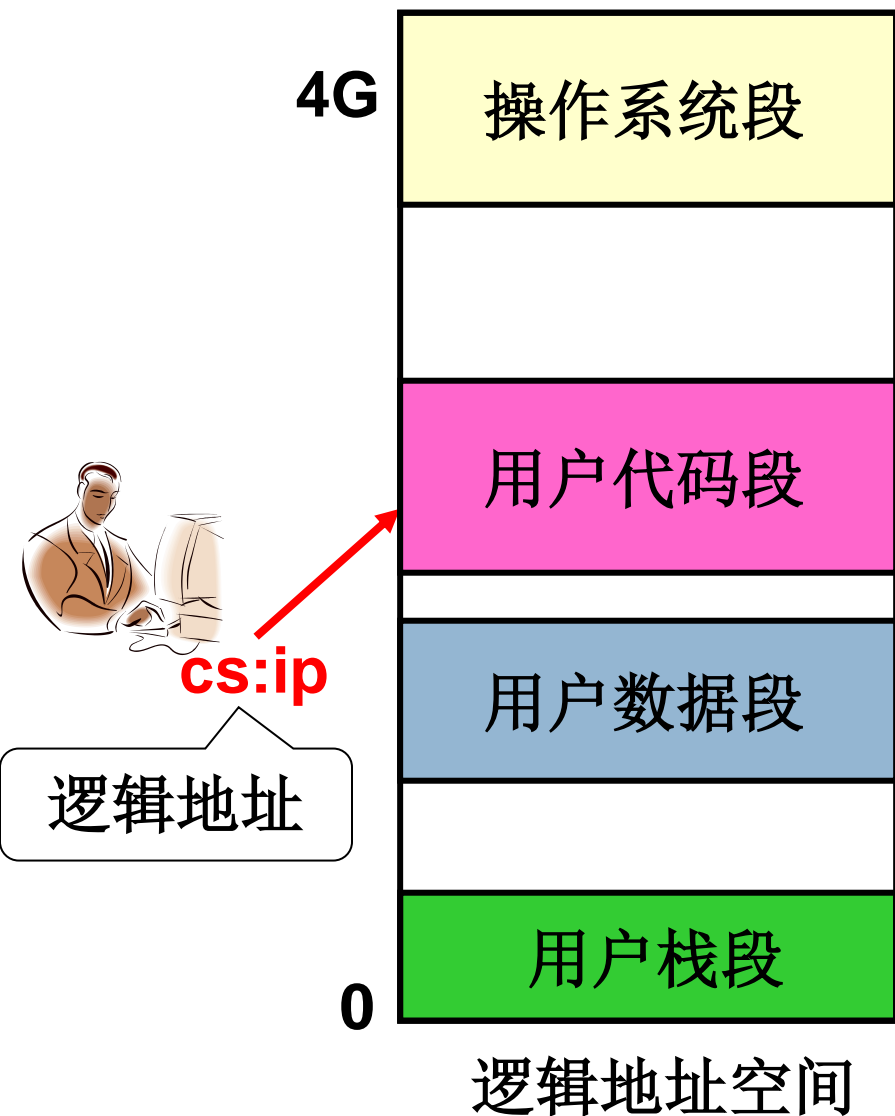


# 内存管理视图





# 用户眼里的内存!

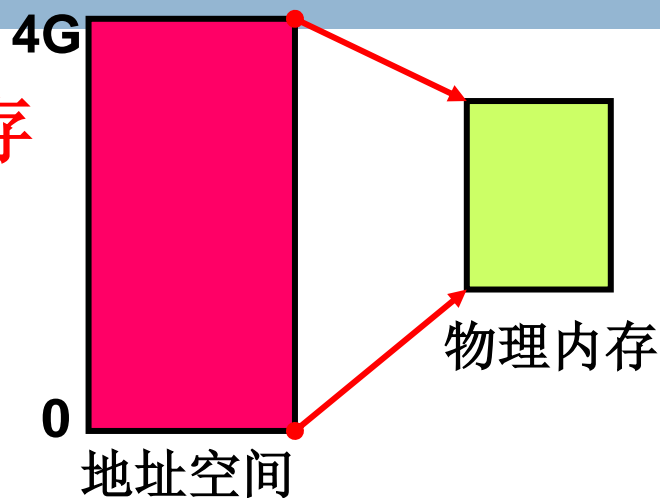


- 1个**4GB**(很大)的地址空间
  - 用户可随意使用该地址空间，就象单独拥有**4G**内存
  - 这个地址空间怎么映射到物理内存，用户全然不知
- 必须映射，否则不能用!
- 这种内存管理和使用方式被称为“**虚拟内存**”

# 虚拟内存的优点

## ■ 优点1: 程序地址空间 > 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

# 什么是虚拟内存?

## 什么是虚拟内存?

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。

## 9.2 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

部分逻辑地址对应段表项，发现缺段后调入

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



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

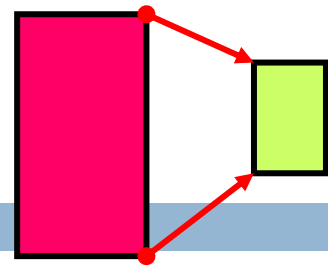
分页易于硬件实现、对用户透明，适合请求调入

物理页号

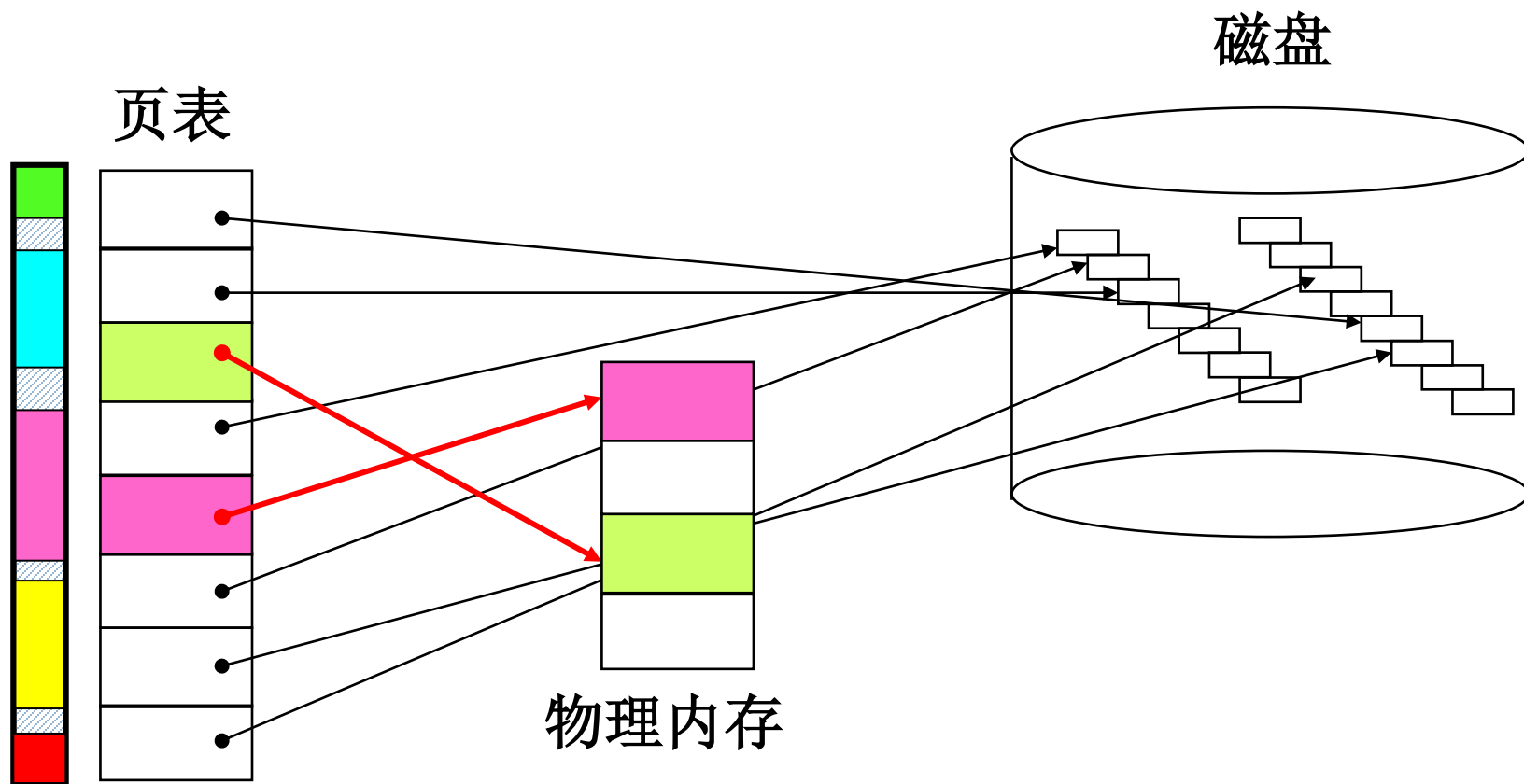
偏移

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，其它页呢？



# 如何记录页是否在内存?

## 某些页不在内存中的页表

帧号    有效/无效位

	V
	V
	V
	V
	i
....	
	i
	i

改造后的页表

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

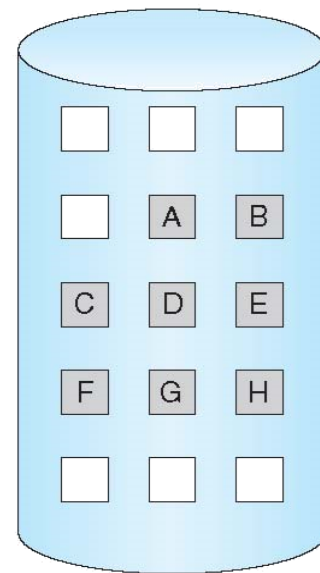
逻辑内存

有效-无效位 valid-invalid bit	
帧号	
0	4 v
1	i
2	6 v
3	i
4	i
5	9 v
6	i
7	i

页表

0
1
2
3
4
5
6
7
8
9
10
11
12
13

物理内存



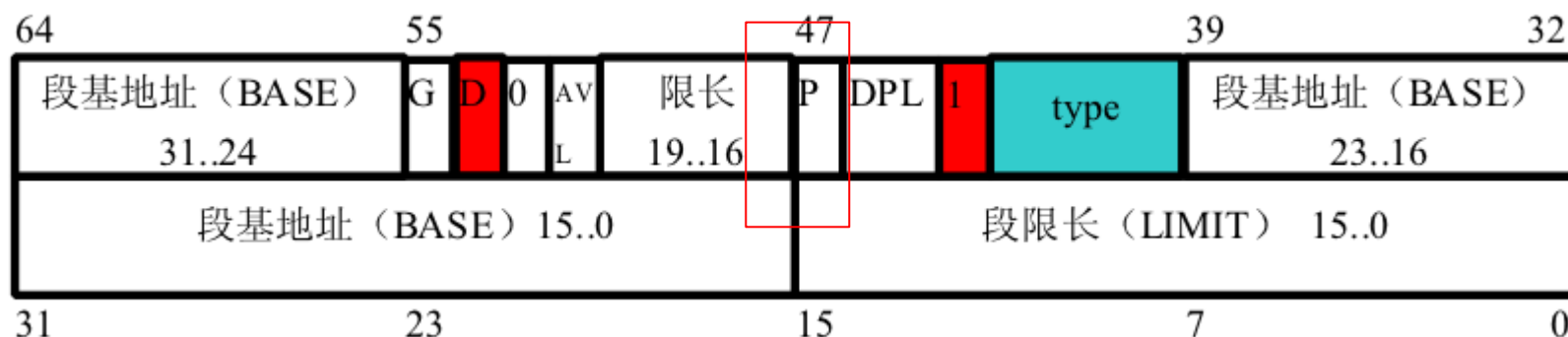
辅存 (磁盘)

- 改造页表，页表项增加“有效/无效位”



# 回忆: Intel x86的分段硬件 – LDT、GDT

## ■ 段描述符: LDT(GDT)中的表项



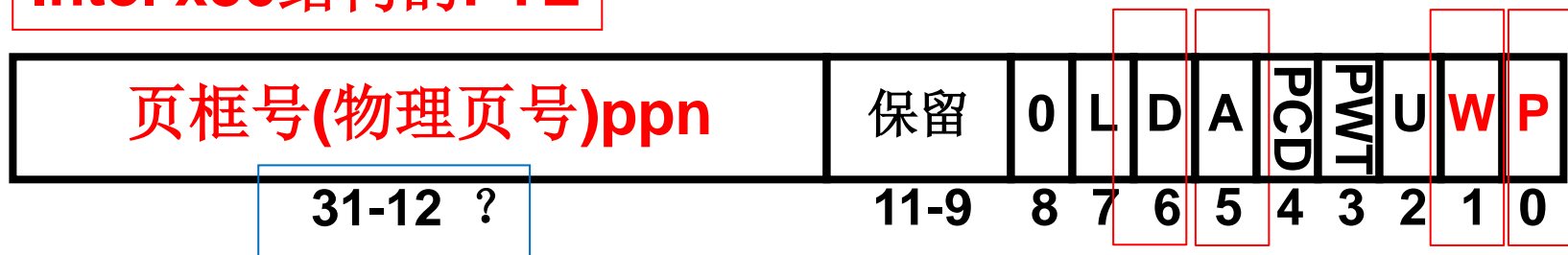
十进制值	TYPE				说明
		E	W	A	
0	0	0	0	0	只读
1	0	0	0	1	只读、已访问
2	0	0	1	0	读写
3	0	0	1	1	读写、已访问
4	0	1	0	0	只读、向下扩展
5	0	1	0	1	只读、向下扩展、已访问
6	0	1	1	0	读写、向下扩展
7	0	1	1	1	读写、向下扩展、已访问

十进制值	TYPE				说明
		C	R	A	
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

# 回忆：Intel x86的分页硬件

## Intel x86结构的PTE



P--位0是存在（Present）标志：按需调页

A--位5是已访问（Accessed）标志：最近最少访问可以被换出

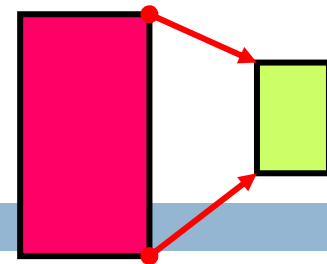
D--位6是页面已被修改（Dirty）标志：是否需要换出？未被改写重新加载即可，  
一个程序创建多进程时复制。

R/W--位1是读/写（Read/Write）标志：读写权限（数据、代码）

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...

显然是一个很好理解的过程

**load [addr]**

但完成这个过程很费时间(特殊时候一条指令会引起几次调页)!

页表

页错误处理程序

磁盘

物理内存

(1)

(2)

(3)

(6)

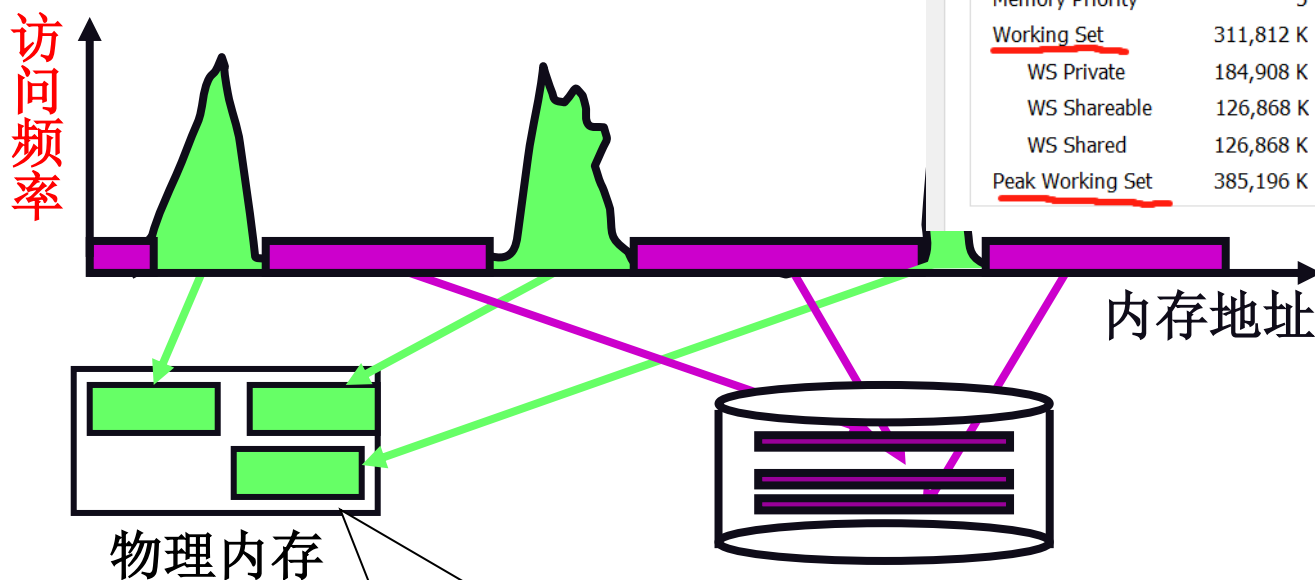
(5)

(4)

# 请求调页为什么可行?

## ■ 分配给一个进程的物理页

## ■ 如何设计这些参数? 再从计算



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!

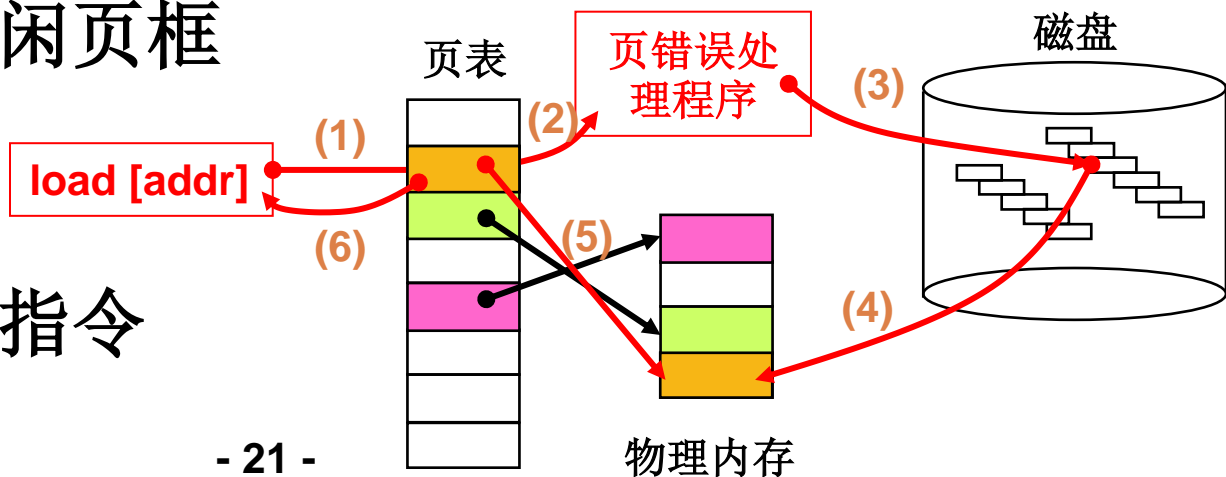
TCP/IP	Security	Environment	Job	Strings
Image	Performance	Performance Graph	GPU Graph	Threads
CPU		I/O		
Priority	8	I/O Priority	Normal	
Kernel Time	0:01:00.296	Reads	38,770	
User Time	0:01:04.343	Read Delta	0	
Total Time	0:02:04.640	Read Bytes Delta	0	
Cycles	235,657,340,626	Writes	6,148	
Virtual Memory		Write Delta	0	
Private Bytes	230,156 K	Write Bytes Delta	0	
Peak Private Bytes	280,696 K	Other	59,875	
Virtual Size	5,349,916 K	Other Delta	0	
Page Faults	501,128	Other Bytes Delta	0	
Page Fault Delta	0	Handles		
Physical Memory		Handles	3,266	
Memory Priority	5	Peak Handles	3,266	
Working Set	311,812 K	GDI Handles	365	
WS Private	184,908 K	USER Handles	127	
WS Shareable	126,868 K			
WS Shared	126,868 K			
Peak Working Set	385,196 K			

# 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**)

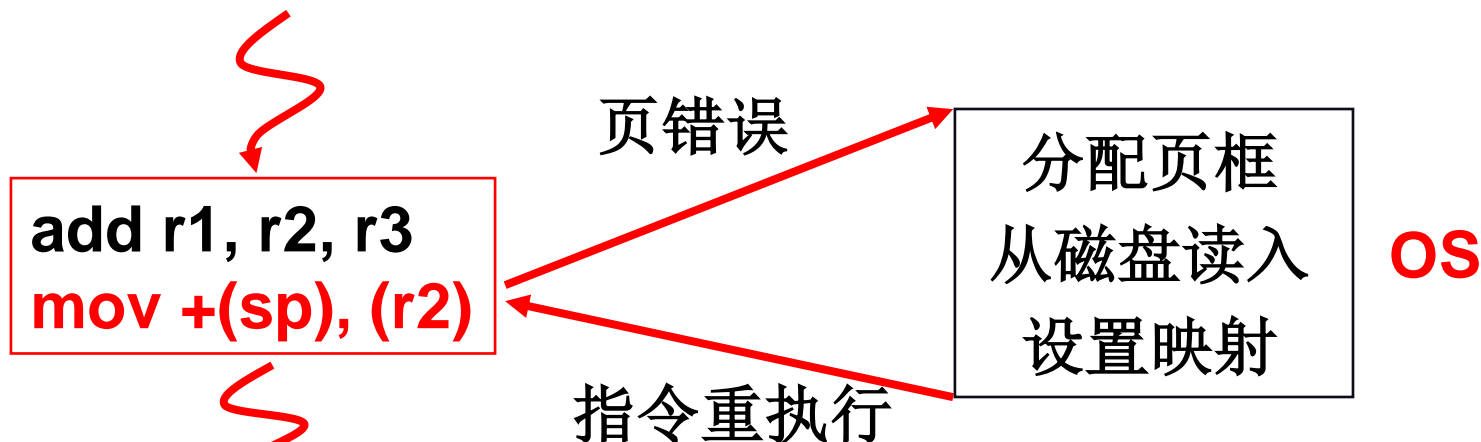
学过磁盘处理后  
自然就明白了!

- **(2):** OS启动时设置“缺页中断”
- **(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 `load [addr]`
- **(6):** 重新开始指令

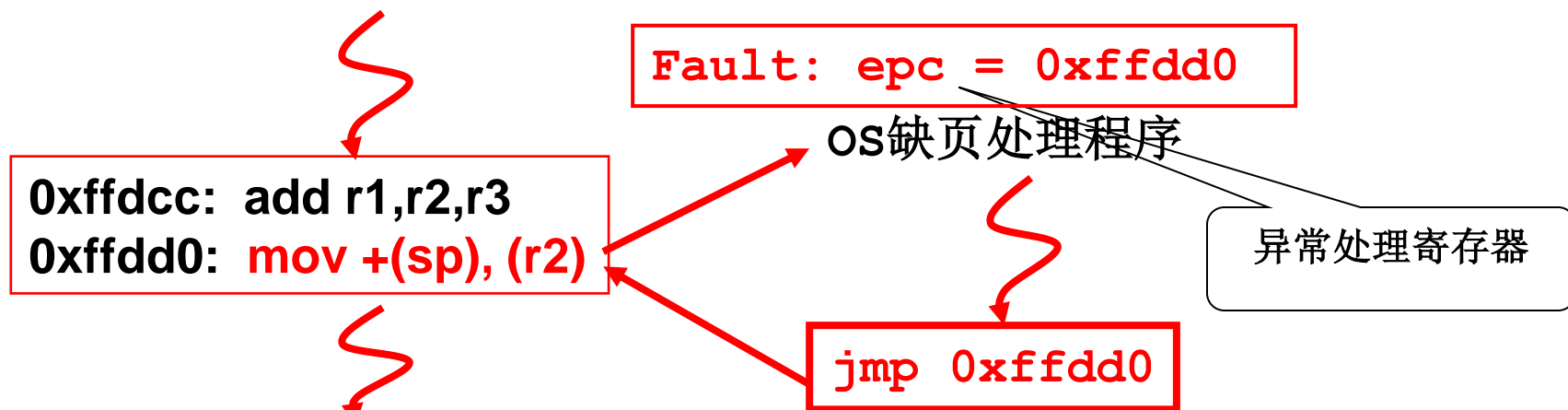


# 如何重新开始指令?

- 在指令执行过程中出现页错误



- 显然这一切应该对用户透明! 需要一点硬件支持



# 如何选一个空闲页框?

## 页面淘汰(置换)

- 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...

## 9.3 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

■ **FIFO**算法:先入先出, 即淘汰最早调入的页面

■ 实例: 进程分配了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:	A	B	C	A	B	D	A	D	B	C	B
Page:											
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

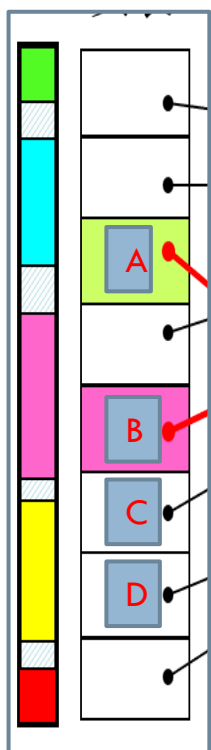
■ 选**A、B、C**中未来最远将使用的, 是理想办法!

# OPT(MIN)页面置换

- **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**



访问 页框	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的可实现性)

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**

访问 页框	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

- 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, 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, 选则计数值最小的页。

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

	A	B	C	A	B	D	A	D	B	C	B
<b>A</b>	1	1	1	4	4	4	7	7	7	7	7
<b>B</b>	0	2	2	2	5	5	5	5	9	9	11
<b>C</b>	0	0	3	3	3	3	3	3	3	10	10
<b>D</b>	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!

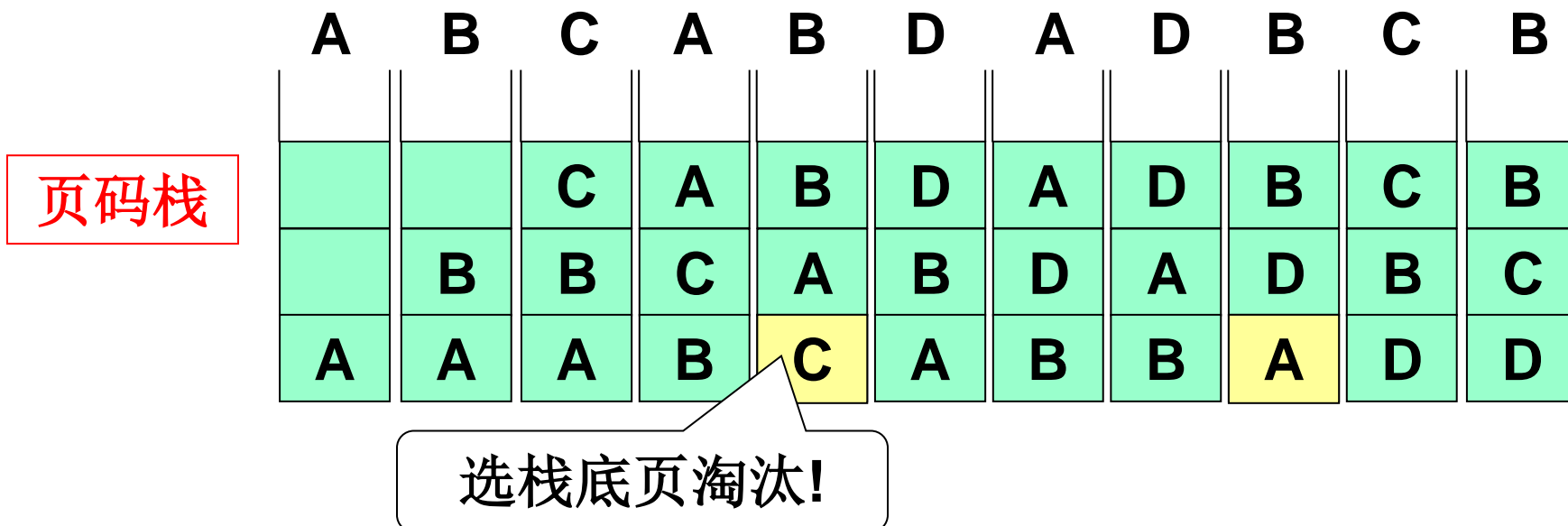
- 每次地址访问都需要修改时间戳, 需维护一个全局时钟 (该时钟溢出怎么办?), **需要找到最小值** ... 这样的实现代价较大 ⇒ **几乎没人用**

# LRU准确实现方法2:页码栈法

## ■ 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**



■ 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（访问）位(referenc

再给一次机会  
(Second  
Chance

## Intel x86结构的PTE



P--位0是存在（Present）标志：按需调页

A--位5是已访问（Accessed）标志：最近最少访问可以被换出

D--位6是页面已被修改（Dirty）标志：是否需要换出？未被改写重新加载即可，一个程序创建多进程写时复制。

R/W--位1是读/写（Read/Write）标志：读写权限（数据、代码）

换出到SWAP分区后地址存到哪里？

可以存到页框号的位置

# 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**（定时可否？）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。

访问 页框	A	B	C	A	B	D	A	D	B	C	B
1	A*	A*	<del>A*</del>	<del>A*</del>	<del>A*</del>	D*	D*	D*	<del>D*</del>	C*	C*
2	<del>A*</del>	B*	B*	B*	B*	<del>B*</del>	A*	A*	A*	<del>A*</del>	<del>A*</del>
3		<del>B*</del>	C*	C*	C*	C	<del>C*</del>	<del>C*</del>	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**

# Clock算法分析的改造

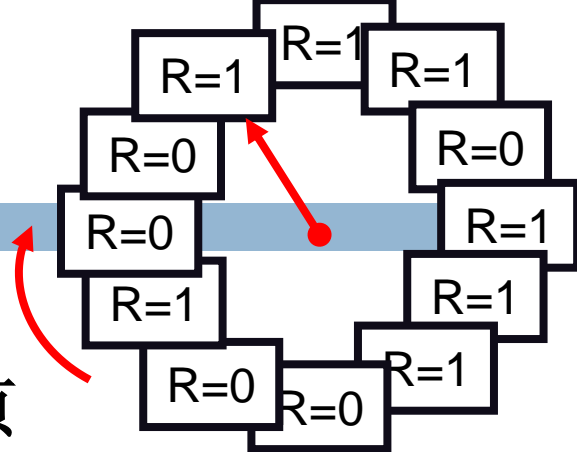
- 如果缺页很少，结果？**所有的R=1**
  - hand scan一圈后淘汰当前页，将调入页插入hand位置，hand前移一位 **退化为FIFO!**
  - 原因：记录了太长的历史信息... 怎么办？
  - 定时清除R位... **再来一个扫描指针!**

指针2：用来清除R位，移动速度要快!

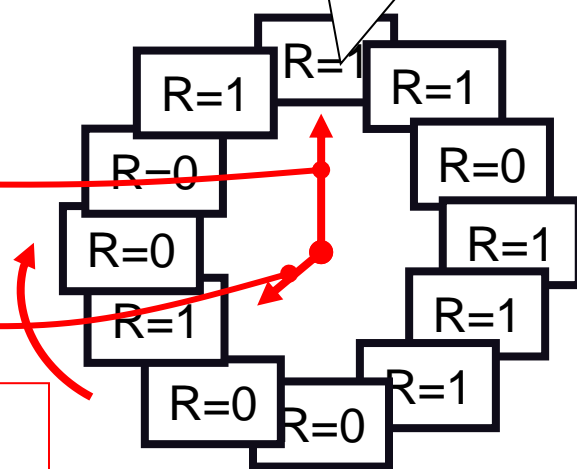
指针1：用来选择淘汰页，移动速度慢!

**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!**



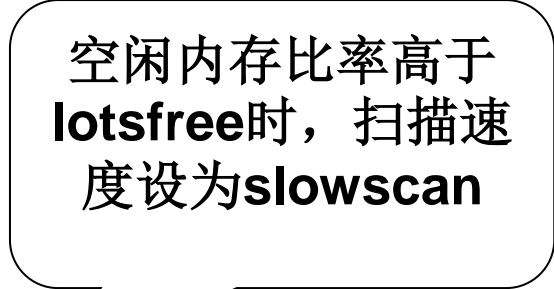
更像  
**Clock**吧!





## ■ 设定值吗？ 系统负载并不固定...

A diagram of a 2D lattice structure. The lattice consists of several rectangular sites, each labeled with either  $R=0$  or  $R=1$ . The sites are arranged in a staggered pattern. A horizontal blue band passes through the middle of the lattice. Two red arrows originate from a central red dot: one points vertically upwards and the other points diagonally downwards and to the left. A curved red arrow on the left side of the lattice indicates a counter-clockwise direction.



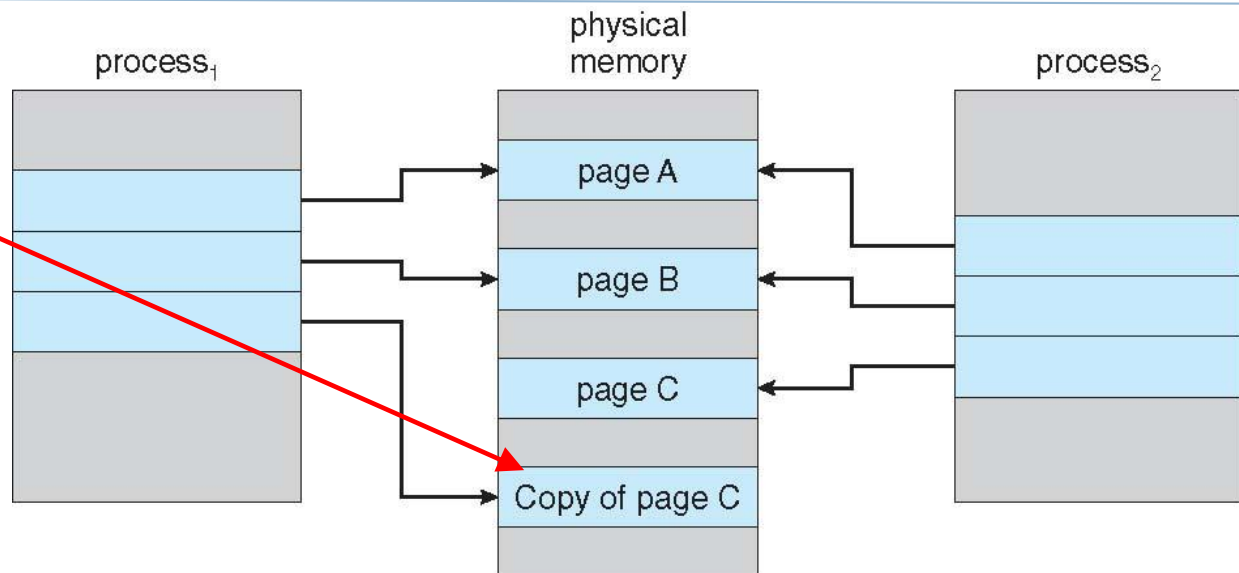
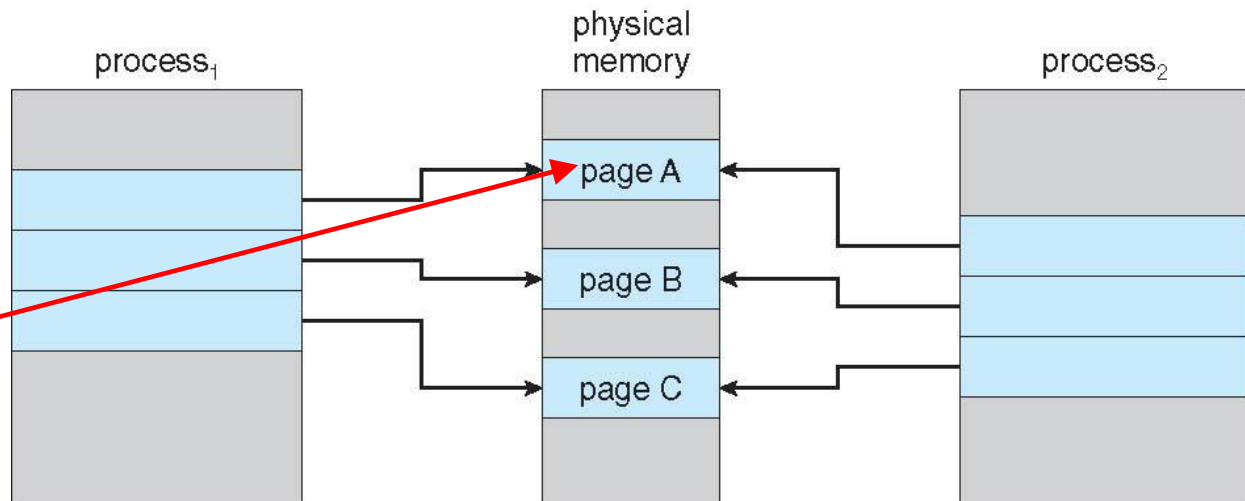


## 9.4 其他相关问题

- (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

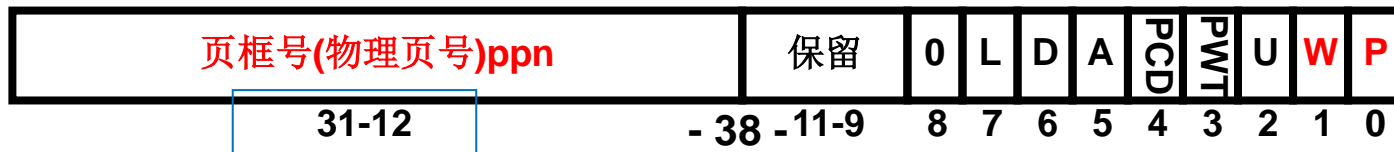
# 交换空间--页面置换到什么地方

请求调页系统中的外存分为：存放（可执行）文件的**文件区**（原始存放位置），存放交换页面的**交换区**。

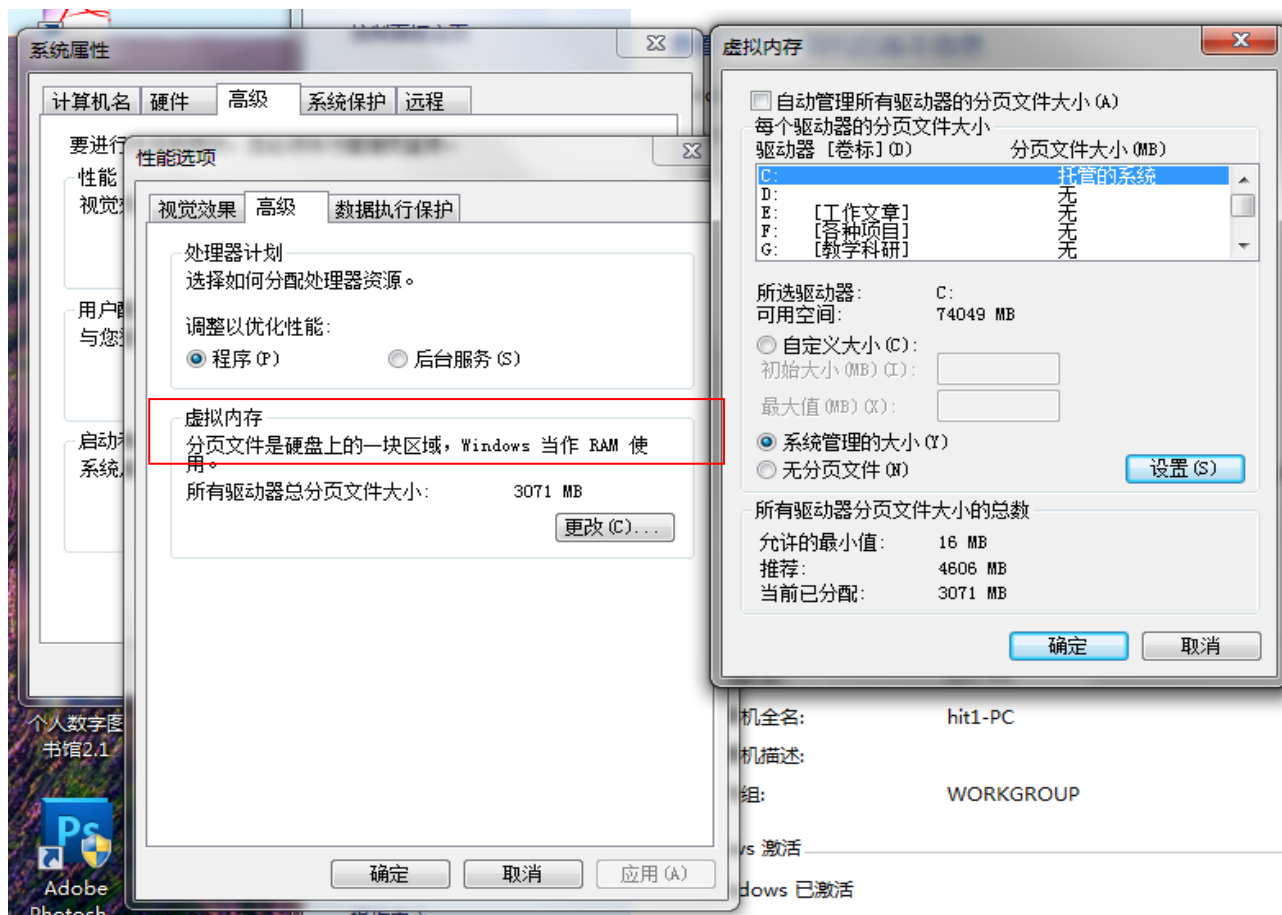
- **系统拥有足够的交换区空间**：可以全部从交换区调入所需页面，以提高调页速度。为此，在进程运行前，需将与该进程有关的文件从文件区复制到交换区。
- **系统缺少足够的交换区空间**：凡不会被修改的文件都直接从文件区调入；而当换出这些页面时，由于它们未被修改而不必再将它们换出。对于可能被修改的部分，将它们换出时须放到交换区，需要时再从交换区调入。
- **UNIX方式**：与进程有关的原始文件都放在文件区，故未运行过的页面，都应从文件区调入。曾经运行过但又被换出的页面，被放在交换区，下次调入时从交换区调入。进程请求的**共享页面**若被其他进程调入内存，则无需再从对换区调入。

进程间共享内存如何实现？

Intel x86结构的PTE



# 交换空间--页面置换到什么地方



windows中pagefile.sys文件与存放位置

固态硬盘如何发挥作用?

# 工作集

**工作集（驻留集）**：是给进程分配的物理内存空间，其一般是动态变化的。

■ 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：

- a) 分配给一个进程的物理内存越小，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) 如果一个进程分配的物理内存（页框）过少，尽管有局部性原理，页错误率仍然会相对较高。
- c) 如驻留集较大，由于局部性原理，当给特定进程分配更多的主存空间时，对进程的缺页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: 1) 各进程平均分配, 2) 根据程序大小按比例分配, 3) 按优先权分配。
- 可变分配(variable-allocation): 工作集大小可动态变化, 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

The screenshot shows the Windows Task Manager interface with the 'Processes' tab selected. A 'Select columns' dialog box is open, allowing users to choose which columns to display in the process list. The dialog has a list of columns with checkboxes next to them. The 'CPU' column is already selected in the Task Manager interface. The 'Select columns' dialog shows the following options:

- ☐ PID (进程标识符)
- ☒ 用户名
- ☐ 会话 ID
- ☒ CPU 使用率
- ☐ CPU 时间
- ☐ 内存 - 工作集
- ☐ 内存 - 高峰工作集
- ☐ 内存 - 工作集增量
- ☒ 内存 - 专用工作集
- ☐ 内存 - 提交大小
- ☐ 内存 - 页面缓冲池
- ☐ 内存 - 非页面缓冲池
- ☐ 页面错误
- ☐ 页面错误增量
- ☐ 基本优先级

The Task Manager process list shows the following columns: 映像名称 (Image Name), 用户名 (User Name), CPU, 内存 (专用工作集) (Memory (Private Working Set)), and 描述 (Description). The status bar at the bottom shows: 进程数: 59 (Number of Processes: 59), CPU 使用率: 61% (CPU Usage: 61%), 物理内存: 71% (Physical Memory: 71%).

On the left side of the image, there is a vertical list of memory-related terms with green boxes highlighting some of them:

- CPU使用率
- CPU时间
- 内存-工作集
- 内存-高峰工作集
- 内存-工作集增量
- 内存-专用工作集
- 内存-提交大小

On the right side, there is a table showing memory usage for various processes:

映像名称	内存 (专用工作集)	线程数
QQ.exe	118,452 K	8
QQExternal.exe	22,760 K	10
taskmgr.exe	6,684 K	6
QQExternal.exe	8,560 K	8
dwm.exe	20,988 K	8
ktcentr.exe	1,552 K	44
ieexplore.exe	46,684 K	37
wisptis.exe	1,536 K	8
ieexplore.exe	127,932 K	13
ieexplore.exe	91,568 K	2
taskeng.exe	1,316 K	1
WINWORD.EXE	39,792 K	10
FlashUtil32.exe	3,536 K	12
ieexplore.exe	57,276 K	5
ieexplore.exe	85,524 K	26
POWERPNT.EXE	200,796 K	9
wuaclt.exe	824 K	3
kphonetray.exe	38,228 K	11
kvipwiz.exe	2,912 K	1
ieexplore.exe	29,008 K	4
ieexplore.exe	49,088 K	2
windowmsgserver32	512 K	4

API: SetProcessWorkingSetSize

# linux工作集

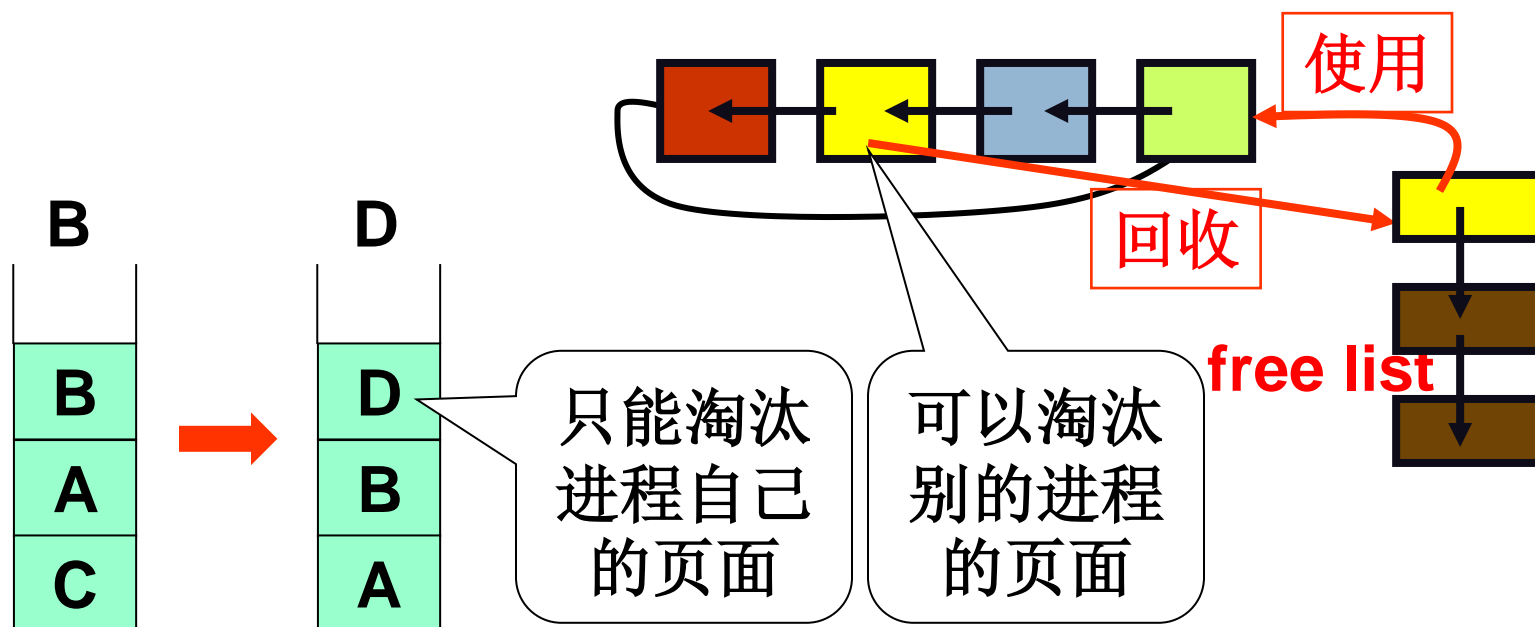
- ps命令是进程查看命令
- RSS是进程使用的驻留集大小或者是实际内存的大小， Kbytes字节。

```
chengjie@utuntu:~/oslab$ ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.4	183160	8064	?	Ss	01:48	0:14	/sbin/init sp
root	2	0.0	0.0	0	0	?	S	01:48	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	I<	01:48	0:00	[rcu_gp]

# 两种置换策略

## 全局置换 局部置换



- 全局置换: 实现简单、有利于系统整体资源使用
- 但全局置换也需要考虑: 有些进程本来分配的物理内存就很少。

# 局部置换需要考虑的关键问题

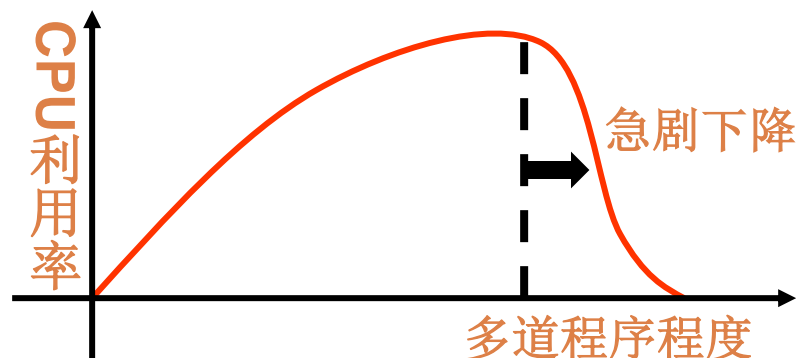
## ■ 给进程分配多少页框(帧frame)

- 分配的多，请求调页的意义就没了！ 尽量少一些？
- 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？

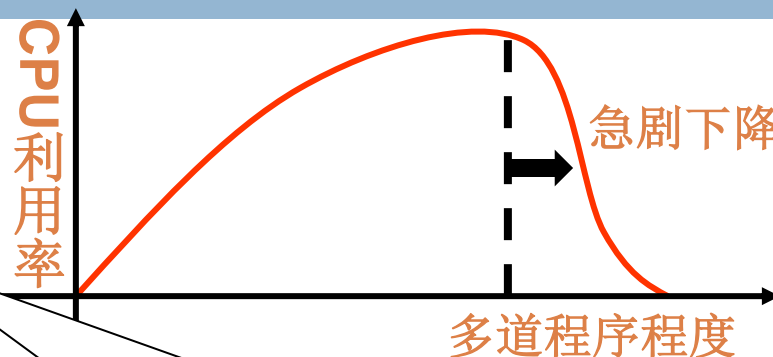


# CPU利用率急剧下降的原因

- 系统内进程增多  $\Rightarrow$  每个进程的缺页率增大  $\Rightarrow$  缺页率增大到一定程度，进程总等待调页完成  $\Rightarrow$  CPU利用率降低  $\Rightarrow$  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？

# Belady异常

## ■ 来看一个例子!

■ 页引用序列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
	2	2	2	1	1	1		3	3
		3	3	3	2	2		2	4

4frame

10faults

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

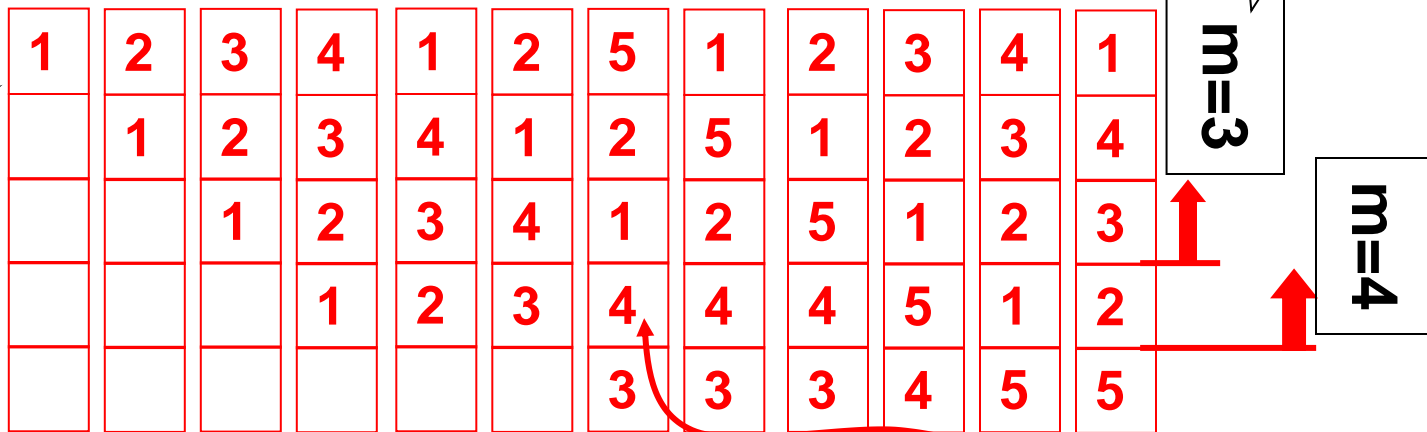
# 什么样的页置换没有Belady异常

m是分配的  
的帧数

看个模型!

引用序列1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

LRU栈实现

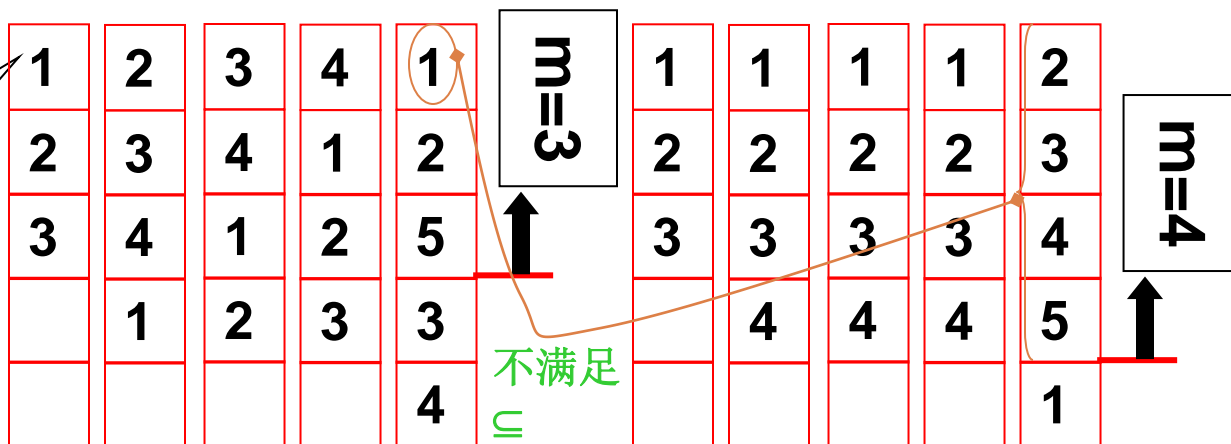


m帧数, r  
引用的页  
集合

特征:  $M(m, r) \subseteq M(m+1, r)$ , 如  $\{5, 2, 1\} \subseteq \{5, 2, 1, 4\}$  ( $m=3$ )

满足这一特征的算法称为栈式算法!

看看  
FIFO



结论: 栈式算法无Belady异常, LRU属于栈式算法!  
赛跑, 前四名肯定包含前三名。

# Linux进程页面置换控制

- Linux通过一个参数swappiness来控制，涉及到复杂的算法。
  - ▣ swappiness可为 0-100，控制进程页换出的优先级（swap 的使用程度）。
  - ▣ 高数值，在进程不活跃时更容易被换出到swap分区。
  - ▣ 低数值，在进程不活跃时不容易被换出到swap分区。
  - ▣ 默认值 60，注意：这个只是一个权值，不是一个百分比值，涉及到系统内核复杂的算法。

如果内存足够大，应当告诉 linux 不必太多的使用 SWAP 分区。

- ① swappiness=0的时候表示最大限度使用物理内存，然后才是swap空间
- ② swappiness = 100的时候表示积极的使用swap分区，并且把内存上的数据及时的搬运到swap空间里面。



# Linux进程页面置换控制

## □ Linux通过一个参数swappiness来控制

涉及到复杂的

现在一般1个G的内存可修改为10，2个G的可改为5，甚至是0。具体这样做：

□ swappiness可为 0-100 (0表示不用swap，100表示尽量用swap)

□ 高数值，在进程不活跃时

□ 低数值，在进程不活跃时

□ 默认值 60，注意：这涉及到系统内核复杂的算

1.查看你的系统里面的swappiness

```
$ cat /proc/sys/vm/swappiness
```

不出意外的话，你应该看到是 60

2.修改swappiness值为10

```
$ sudo sysctl vm.swappiness=10
```

但是这只是临时性的修改，在你重启系统后会恢复默认的60，为

长治久安，还要更进一步：

```
$ sudo gedit /etc/sysctl.conf
```

在这个文档的最后加上这样一行：

```
vm.swappiness=10
```

如果内存足够大，应当告诉 linux 不必太多的使用 SWAP 分区。

① swappiness=0的时候表示最大限度使用物理内存，然后才是swap空间

② swappiness=100的时候表示积极的使用swap分区，并且把内存上的数据及时的搬运到swap空间里面。

# 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

内存利用率高,  
响应时间快...

why?

## ■ 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  $\Rightarrow$  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  $\Rightarrow$  内存利用率高
- 可编写比内存大的程序  $\Rightarrow$  使用一个大地址空间(虚拟内存)
- 部分程序在内存  $\Rightarrow$  其他部分在磁盘  $\Rightarrow$  需要的时候调入内存
- 页表项存在P位  $\Rightarrow$  缺页产生中断  $\Rightarrow$  中断处理完成页面调入
- 调入页面需要一个空闲页框  $\Rightarrow$  如果没有空闲页框  $\Rightarrow$  置换
- 置换方法（页表项存在A位）  $\Rightarrow$  FIFO  $\rightarrow$  MIN  $\rightarrow$  LRU  $\rightarrow$  Clock
- 需要给进程分配页框  $\Rightarrow$  全局、局部  $\Rightarrow$  颠簸  $\Rightarrow$  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