

置换算法的功能和目标

■ 功能

- ▶ 当出现缺页异常，需调入新页面而内存已满时，置换算法选择被置换的物理页面

■ 设计目标

- ▶ 尽可能减少页面的调入调出次数
- ▶ 把未来不再访问或短期内不访问的页面调出

■ 页面锁定(frame locking)

- ▶ 描述必须常驻内存的逻辑页面
- ▶ 操作系统的关键部分
- ▶ 要求响应速度的代码和数据
- ▶ 页表中的锁定标志位(lock bit)

置换算法的评价方法

- 记录进程访问内存的页面轨迹

- ▶ 举例: 虚拟地址访问用(页号, 位移)表示

(3,0), (1,9), (4,1), (2,1), (5,3), (2,0), (1,9),
(2,4), (3,1), (4,8)

- ▶ 对应的页面轨迹

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

替换如 c, a, d, b, e, b, a, b, c, d

- 评价方法

- ▶ 模拟页面置换行为, 记录产生缺页的次数

- ▶ 更少的缺页, 更好的性能

页面置换算法分类

■ 局部页面置换算法

- ▶ 置换页面的选择范围仅限于当前进程占用的物理页面内
- ▶ 最优算法、先进先出算法、最近最久未使用算法
- ▶ 时钟算法、最不常用算法

■ 全局页面置换算法

- ▶ 置换页面的选择范围是所有可换出的物理页面
- ▶ 工作集算法、缺页率算法



操作系统

Operating System



操作系统

Operating System

最优页面置换算法(OPT, optimal)

- 基本思路
- 算法实现 未来最长时间不访问的页面
- 算法特点 计算内存中每个逻辑页面的下一次访问时间
 - ▶ 替换未来最长时间不访问的页面
 - ▶ 实际系统中无法实现
 - ▶ 无法预知每个页面在下次访问前的等待时间
 - ▶ 作为置换算法的性能评价依据
 - ▶ 在模拟器上运行某个程序，并记录每一次的页面访问情况
 - ▶ 第二遍运行时使用最优算法

最优页面置换算法示例

时间		0	1	2	3	4	5	6	7	8	9	10
访问请求			c	a	d	b	e	b	a	b	c	d
物理 帧号	0	a	a	a	a	a	a	a	a	a	a	
	1	b	b	b	b	b	b	b	b	b	b	
	2	c	c	c	c	c	c	c	c	c	c	
	3	d	d	d	d	d	e	e	e	e	e	
缺页状态							●					●
每页的下次 访问时间							a=7 b=6 c=9 d=10					a=? b=? c=? d=?

先进先出算法 (First-In First-Out, FIFO)

■ 思路

- ▶ 选择在内存驻留时间最长的页面进行置换

■ 实现

- ▶ 维护一个记录所有位于内存中的逻辑页面链表
- ▶ 链表元素按驻留内存的时间排序，链首最长，链尾最短
- ▶ 出现缺页时，选择链首页面进行置换，新页面加到链尾

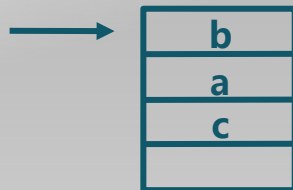
■ 特征

- ▶ 实现简单
- ▶ 性能较差，调出的页面可能是经常访问的
- ▶ 进程分配物理页面数增加时，缺页并不一定减少(Belady现象)
- ▶ 很少单独使用

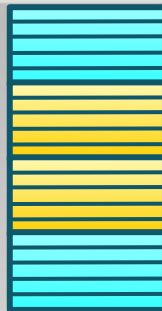
FIFO

在4个页帧中执行:

▣ 假定初始a -> b -> c -> d顺序



访问页面链表



进程占用
物理内存

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求		c	a	d	b	e	b	a	b	c	d
物理帧号	0	a	a	a	a	e	e	e	e	e	d
	1	b	b	b	b	b	b	a	a	a	a
	2	c	c	c	c	e	c	c	b	b	b
	3	d	d	d	d	d	d	d	d	c	c
缺页状态						●		●	●	●	●

最近最久未使用算法 (Least Recently Used, LRU)

■ 思路

- ▣ 选择**最长时间没有被引用的**页面进行置换
- ▣ 如某些页面长时间未被访问，则它们在将来还可能会长时间不会访问

■ 实现

- ▣ 缺页时，计算内存中每个逻辑页面的**上一次**访问时间
- ▣ 选择**上一次使用到当前时间最长的**页面

■ 特征

- ▣ 最优置换算法的一种近似

最近最未被使用算法(LRU)

置换的页面是最长时间没有被引用的

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求		c	a	d	b	e	b	a	b	c	d
物理帧号	0	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c → e	e	e	e	e	e → d	
	3	d	d	d	d	d	d	d	d	d → c	c
缺页状态						●				●	●
每页的下次访问时间						a=2 b=4 c=1 d=3				a=7 b=8 e=5 d=3	a=7 b=8 e=5 c=9

LRU算法的可能实现方法

■ 页面链表

- ▣ 系统维护一个按最近一次访问时间排序的页面链表
 - ▣ 链表首节点是最近刚刚使用过的页面
 - ▣ 链表尾节点是最久未使用的页面
- ▣ 访问内存时，找到相应页面，并把它移到链表之首
- ▣ 缺页时，置换链表尾节点的页面

■ 活动页面栈

- ▣ 访问页面时，将此页号压入栈顶，并栈内相同的页号抽出
- ▣ 缺页时，置换栈底的页面

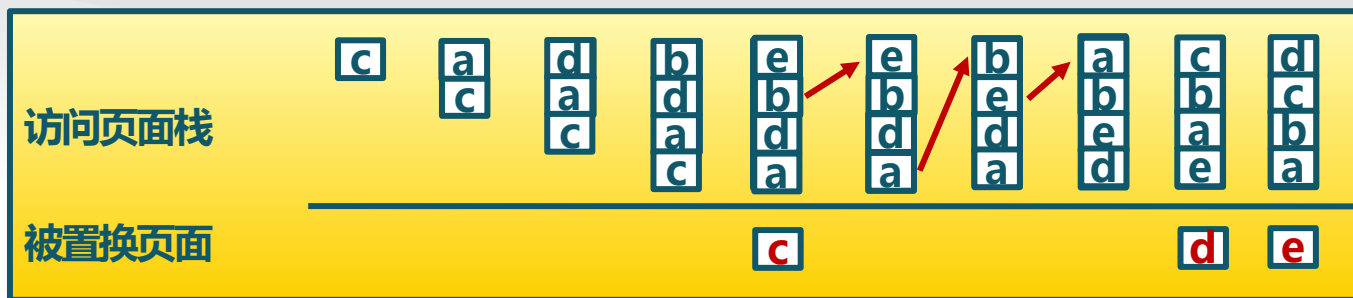
■ 特征

- ▣ 开销比较大

用栈实现LRU算法

保持一个最近使用页面的“栈”

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求		c	a	d	b	e	b	a	b	c	d
物理帧号	0	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	e	e	e	e	e	b
	3	d	d	d	d	d	d	d	d	c	c
缺页状态						●				●	●





操作系统

Operating System

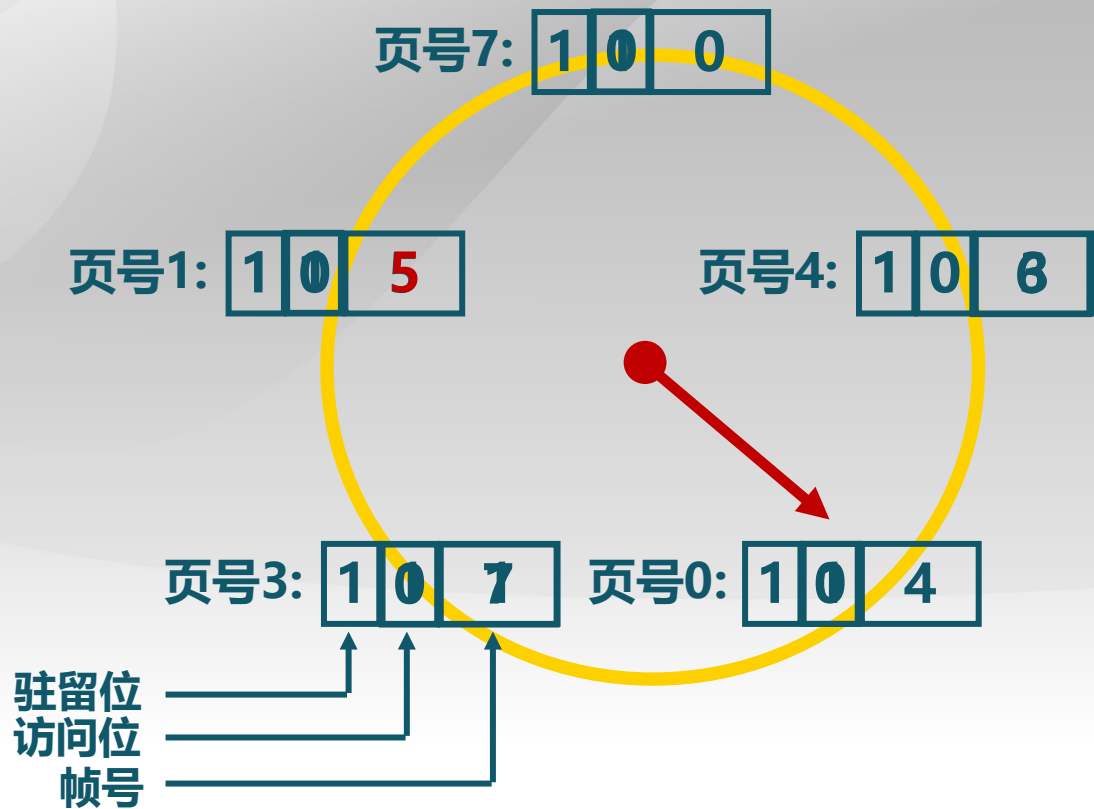
时钟置换算法 (Clock)

- 思路
 - ▣ 仅对页面的访问情况进行大致统计
- 数据结构
 - ▣ 在页表项中增加**访问位**，描述页面在过去一段时间的内访问情况
 - ▣ 各页面组织成**环形链表**
 - ▣ **指针**指向最先调入的页面
- 算法
 - ▣ 访问页面时，在页表项记录页面访问情况
 - ▣ 缺页时，从指针处开始顺序查找未被访问的页面进行置换
- 特征
 - ▣ 时钟算法是LRU和FIFO的折中

时钟置换算法的实现

- 页面装入内存时，访问位初始化为0
- 访问页面（读/写）时，访问位置1
- 缺页时，从指针当前位置顺序检查环形链表
 - ▣ 访问位为0，则置换该页
 - ▣ 访问位为1，则访问位置0，并指针移动到下一个页面，直到找到可置换的页面

时钟置换算法图示



时钟页面置换示例

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求		c	a	d	b	e	b	a	b	c	d
物理帧号	0	a	a	a	a	e	e	e	e	e	d
	1	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	a	a	a	a
	3	d	d	d	d	d	d	d	d	c	c
缺页状态											

驻留页面的页表项	0	a	0	a	1	a	1	a	1	a	1	e	1	e	1	e	1	e	1	d
	0	b	0	b	0	b	0	b	1	b	0	b	1	b	0	b	1	b	0	b
	0	c	1	c	1	c	1	c	1	c	0	c	0	c	1	a	1	a	1	a
	0	d	0	d	0	d	1	d	1	d	0	d	0	d	0	d	0	d	1	c

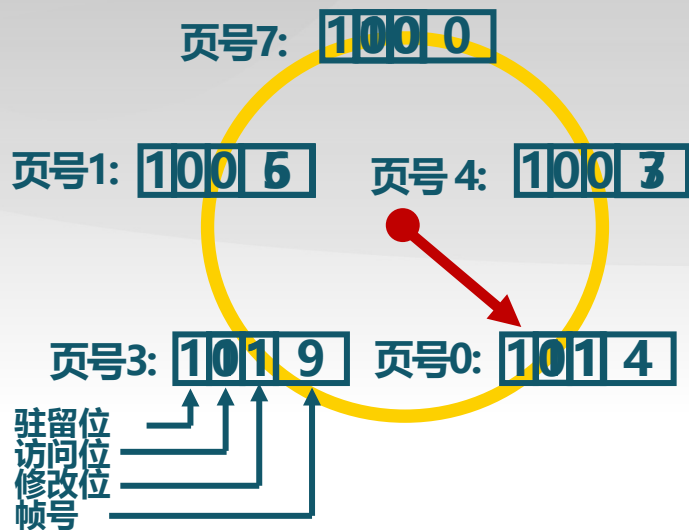
改进的Clock算法

■ 思路

- ▣ 减少修改页的缺页处理开销

■ 算法

- ▣ 在页面中增加修改位，并在访问时进行相应修改
- ▣ 缺页时，修改页面标志位，以跳过有修改的页面



指针扫过前		指针扫过后	
使用位	修改位	使用位	修改位
0	0	置换	
0	1		
1	0		
1	1		
		0	0
		0	0
		0	1

改进的Clock算法

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求		c	a ^w	d	b ^w	e	b	a ^w	b	c	d
物理帧号	0	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	→d
	2	c	c	c	c	→e	e	e	e	e	e
	3	d	d	d	d	d	d	d	d	→c	c
缺页状态						●				●	●

驻留页面的 页表项	00	a	00	a	11	a	11	a	11	a	00	a	00	a	11	a	11	a	11	a	00	a*		
	00	b	00	b	00	b	00	b	11	b	00	b	10	b	10	b	10	b	10	b	10	b	10	d
	00	c	10	c	10	c	10	c	10	c	10	e	10	e	10	e	10	e	10	e	10	e	00	e
	00	d	00	d	00	d	10	d	10	d	00	d	00	d	00	d	00	d	00	d	10	c	00	c

最不常用算法 (Least Frequently Used, LFU)

- 思路
- 实现时，置换访问次数最少的页面
- 为每个页面设置一个访问计数
- 访问页面时，访问计数加1
- LRU和LFU的区别
 - LRU关注多久未访问，时间越短越好
 - LFU关注访问次数，次数越多越好

LFU算法示例

执行在4个页帧中：

▣ 假定最初的访问次数 a- > 8 b- > 5 c- > 6 d- > 2

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求		c ⁷	a ¹	d ¹⁴	b ⁵	e ¹⁸	b ¹	a ¹⁹	b ²⁰	c ²⁰	d ¹⁷
物理帧号	0	a ⁸	a ⁹	a ⁹	a ⁹ → e ¹⁸	e ¹⁸	e ¹⁸	e ¹⁸	e ¹⁸	e ¹⁸ → d ¹⁷	
	1	b ⁵	b ⁵	b ⁵	b ¹⁰	b ¹⁰	b ¹¹ → a ¹⁹	a ¹⁹	a ¹⁹	a ¹⁹	a ¹⁹
	2	c ⁶	c ¹³	c ¹³	c ¹³	c ¹³	c ¹³	c ¹³ → b ²⁰	b ²⁰	b ²⁰	b ²⁰
	3	d ²	d ²	d ¹⁶	d ¹⁶	d ¹⁶	d ¹⁶	d ¹⁶	d ¹⁶ → c ²⁰	c ²⁰	c ²⁰
缺页状态						●		●	●	●	●



操作系统

Operating System

Belady现象

- 现象

- 采用FIFO等算法时，可能出现分配的物理页面数增加，缺页次数反而升高的异常现象

- 原因

- FIFO算法的置换特征与进程访问内存的动态特征矛盾
- 被它置换出去的页面并不一定是进程近期不会访问的

- 思考

- 哪些置换算法没有Belady现象？

FIFO算法有Belady现象

访问顺序: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

物理页面数: 3

缺页次数: 9

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页	1	2	3	4	1	2	5	5	5	3	4	4
头		1	2	3	4	1	2	2	2	5	3	3
头			1	2	3	4	1	1	1	2	5	5
缺页状态	●	●	●	●	●	●	●			●	●	

FIFO算法有Belady现象

访问顺序：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

物理页面数：4

缺页次数：10

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页	1	2	3	4	4	4	5	1	2	3	4	5
头		1	2	3	3	3	4	5	1	2	3	4
头			1	2	2	2	3	4	5	1	2	3
头				1	1	1	2	3	4	5	1	2
缺页状态	●	●	●	●			●	●	●	●	●	●

LRU算法没有Belady 现象

物理页面数: 3 缺页次数: 10

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

物理页面数: 4 缺页次数: 8

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

时钟/改进的时钟页面置换是否有Belady现象?
为什么LRU页面置换算法没有Belady现象?

LRU、FIFO和Clock的比较

- LRU算法和FIFO本质上都是先进先出的思路
 - ▣ LRU依据页面的最近访问时间排序
 - ▣ LRU需要动态地调整顺序
 - ▣ FIFO依据页面进入内存的时间排序
 - ▣ FIFO的页面进入时间是固定不变的
- LRU可退化成FIFO
 - ▣ 如页面进入内存后没有被访问，最近访问时间与进入内存的时间相同
 - ▣ 例如：给进程分配3个物理页面，逻辑页面的访问顺序为1、2、3、4、5、6、1、2、3...

LRU、FIFO和Clock的比较

- LRU算法性能较好，但系统开销较大
- FIFO算法系统开销较小，会发生Belady现象
- Clock算法是它们的折衷
 - ▣ 页面访问时，不动态调整页面在链表中的顺序，仅做标记
 - ▣ 缺页时，再把它移动到链表末尾
- 对于未被访问的页面，Clock和LRU算法的表现一样好
- 对于被访问过的页面，Clock算法不能记录准确访问顺序，而LRU算法可以



操作系统











Operating System

局部置换算法没有考虑进程访存差异

FIFO 页面置换算法: 假设初始顺序 a->b->c

物理页面数: 4

缺页次数: 9

时间	0	1	2	3	4	5	6	7	8	9	10	11	12
访问页面		a	b	c	d	a	b	c	d	a	b	c	d
物理 帧号	0	a	a	a	d	d	d	c	c	c	b	b	b
	1	b	b	b	b	a	a	a	d	d	d	c	c
	2	c	c	c	c	c	b	b	b	a	a	a	d
缺页状态													
缺页状态													

全局置换算法

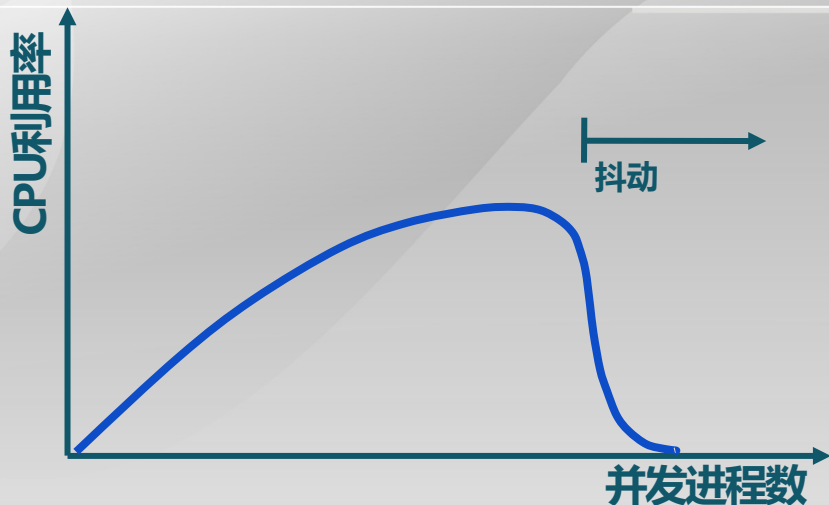
- 思路

- ▶ 全局置换算法为进程分配**可变数目**的物理页面

- 全局置换算法要解决的问题

- ▶ 进程在不同阶段的内存需求是变化的
 - ▶ 分配给进程的内存也需要在不同阶段有所变化
 - ▶ 全局置换算法需要确定分配给进程的物理页面数

CPU利用率与并发进程数的关系



- CPU利用率与并发进程数存在相互促进和制约的关系
 - ▶ 进程数少时，提高并发进程数，可提高CPU利用率
 - ▶ 并发进程导致内存访问增加
 - ▶ 并发进程的内存访问会降低访问的局部性特征
 - ▶ 局部性特征的下降会导致缺页率上升和CPU利用率下降

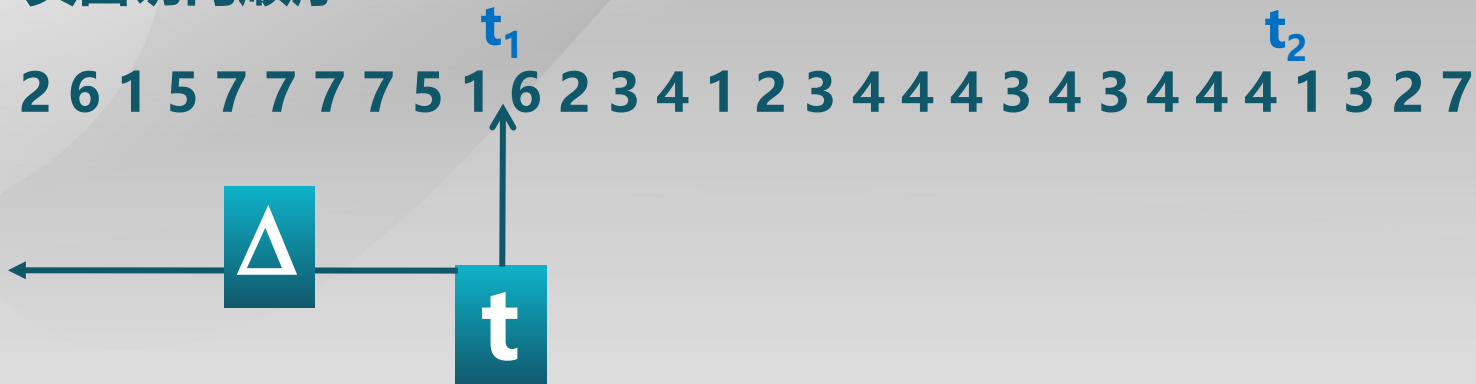
工作集

一个进程当前正在使用的逻辑页面集合，可表示为二元函数 $W(t, \Delta)$

- t 是当前的执行时刻
- Δ 称为工作集窗口 (working-set window) , 即一个定长的页面访问时间窗口
- $W(t, \Delta)$ 是指在当前时刻 t 前的 Δ 时间窗口中的所有访问页面所组成的集合
- $|W(t, \Delta)|$ 指工作集的大小, 即页面数目

进程的工作集示例

页面访问顺序:



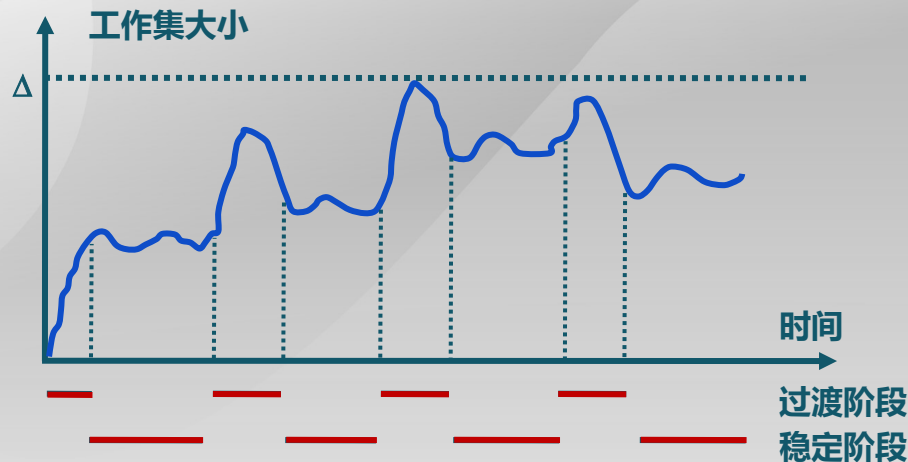
如果 Δ 时间窗口的长度为10, 那么:

$$W(t, \Delta) = \{2, 2, 5, 5, 5, 5, 7\}$$

$$W(t_1, \Delta) = \{1, 2, 5, 6, 7\}$$

$$W(t_2, \Delta) = \{3, 4\}$$

工作集的变化



- 进程开始执行后，随着访问新页面逐步建立较稳定的工作集
- 当内存访问的局部性区域的位置大致稳定时，工作集大小也大致稳定
- 局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值

常驻集

在当前时刻，进程实际驻留在内存当中的页面集合

■ 工作集与常驻集的关系

- ▶ 工作集是进程在运行过程中固有的性质
- ▶ 常驻集取决于系统分配给进程的物理页面数目和页面置换算法

■ 缺页率与常驻集的关系

- ▶ 常驻集 \supseteq 工作集时，缺页较少
- ▶ 工作集发生剧烈变动（过渡）时，缺页较多
- ▶ 进程常驻集大小达到一定数目后，缺页率也不会明显下降

工作集置换算法

■ 思路

- ▣ 换出不在工作集中的页面

■ 窗口大小 τ

- ▣ 当前时刻前 τ 个内存访问的页引用是工作集， τ 被称为窗口大小

■ 实现方法

- ▣ 访存链表：维护窗口内的访存页面链表
- ▣ 访存时，换出不在工作集的页面；更新访存链表
- ▣ 缺页时，换入页面；更新访存链表

工作集置换算法

$\tau = 4$

时间		0	1	2	3	4	5	6	7	8	9	10
访问页面			c	c	d	b	c	e	c	e	a	d
逻辑 页面 状态	页面a	● t=0	●	●	●						●	●
	页面b					●	●	●	●			
	页面c		●	●	●	●	●	●	●	●	●	●
	页面d	● t=-1	●	●	●	●	●					●
	页面e	● t=-2	●					●	●	●	●	●
缺页状态			●			●		●			●	●



操作系统

Operating System

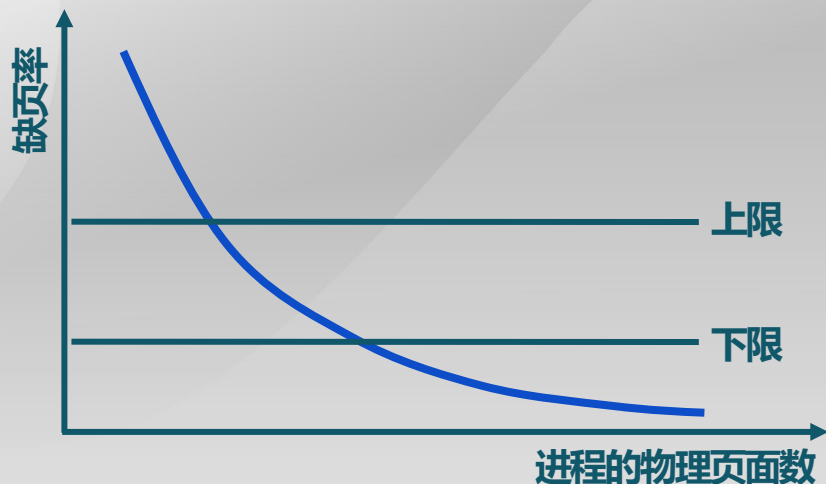
缺页率(page fault rate)

缺页次数 / 内存访问次数 或 缺页平均时间间隔的倒数

■ 影响缺页率的因素

- ▣ 页面置换算法
- ▣ 分配给进程的物理页面数目
- ▣ 页面大小
- ▣ 程序的编写方法

缺页率置换算法 (PFF, Page-Fault-Frequency)



通过调节常驻集大小，使每个进程的缺页率保持在一个合理的范围内

- 若进程缺页率过高，则增加常驻集以分配更多的物理页面
- 若进程缺页率过低，则减少常驻集以减少它的物理页面数

缺页率置换算法的实现

- 访存时，设置引用位标志
- 缺页时，计算从上次缺页时间 t_{last} 到现在 $t_{current}$ 的时间间隔
 - ▣ 如果 $t_{current} - t_{last} > T$ ，则置换所有在 $[t_{last}, t_{current}]$ 时间内没有被引用的页
 - ▣ 如果 $t_{current} - t_{last} \leq T$ ，则增加缺失页到工作集中

缺页率置换算法示例

- 假定窗口大小为 2

时间		0	1	2	3	4	5	6	7	8	9	10
访问页面			c	c	d	b	c	e	c	e	a	d
逻辑 页面 状态	页面a	●	●	●							●	●
	页面b				●	●	●	●	●	●		
	页面c		●	●	●	●	●	●	●	●	●	●
	页面d	●	●	●	●	●						●
	页面e	●	●	●				●	●	●	●	●
缺页状态			●			●		●			●	●
$t_{\text{cur}} - t_{\text{last}}$			1			3		2			3	1



操作系统

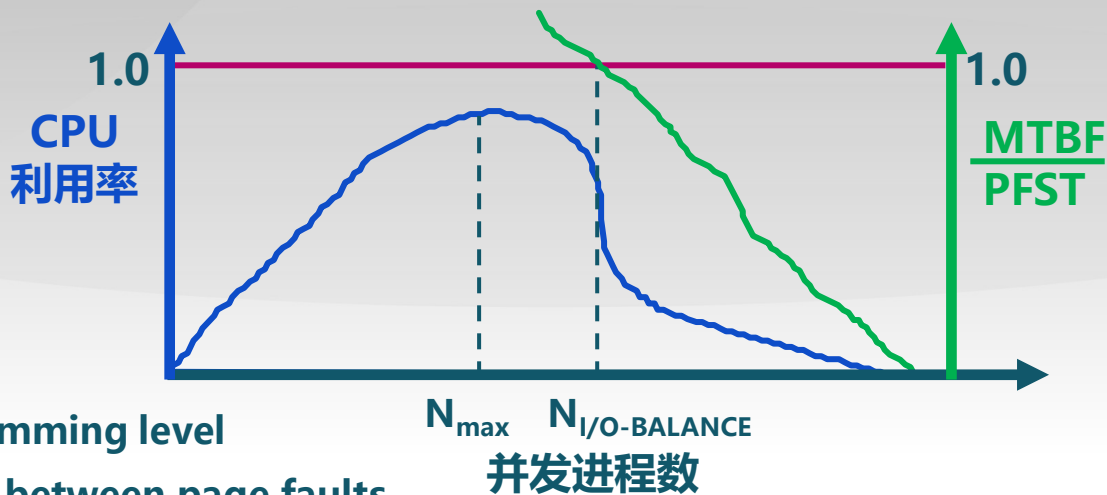
Operating System

抖动问题(thrashing)

- 抖动
 - ▣ 进程物理页面太少，不能包含工作集
 - ▣ 造成大量缺页，频繁置换
 - ▣ 进程运行速度变慢
- 产生抖动的原因
 - ▣ 随着驻留内存的进程数目增加，分配给每个进程的物理页面数不断减小，缺页率不断上升
- 操作系统需在并发水平和缺页率之间达到一个平衡
 - ▣ 选择一个适当的进程数目和进程需要的物理页面数

负载控制

- 通过调节并发进程数 (MPL) 来进行系统负载控制
 - ▣ $\sum WSi$ = 内存的大小
 - ▣ 平均缺页间隔时间(MTBF) = 缺页异常处理时间(PFST)



MPL-multiprogramming level

MTBF-mean time between page faults

PFST-page fault service time



操作系统

Operating System

第七讲虚拟存储：局部页面置换算法

第 5 节页表自映射

向勇、陈渝

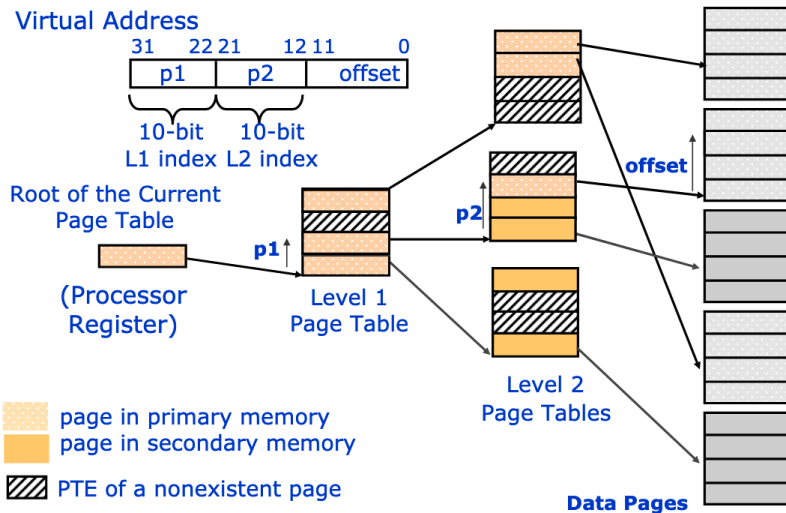
清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

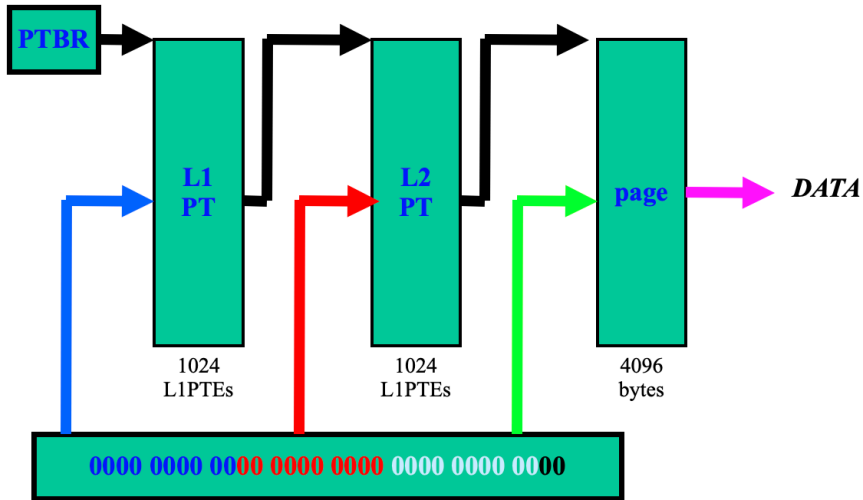
2020 年 5 月 5 日

- 1 第 5 节页表自映射
 - 页表自映射
 - X86-32 页表自映射
 - riscv32 页表自映射

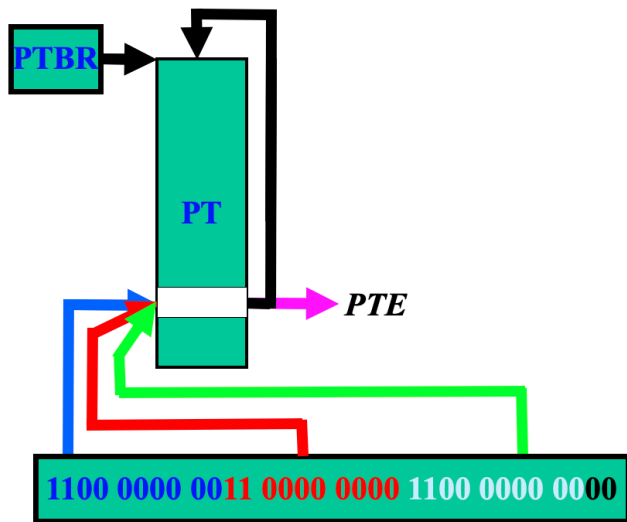
基于 4KB 页面的 32 位 CPU 二级页表



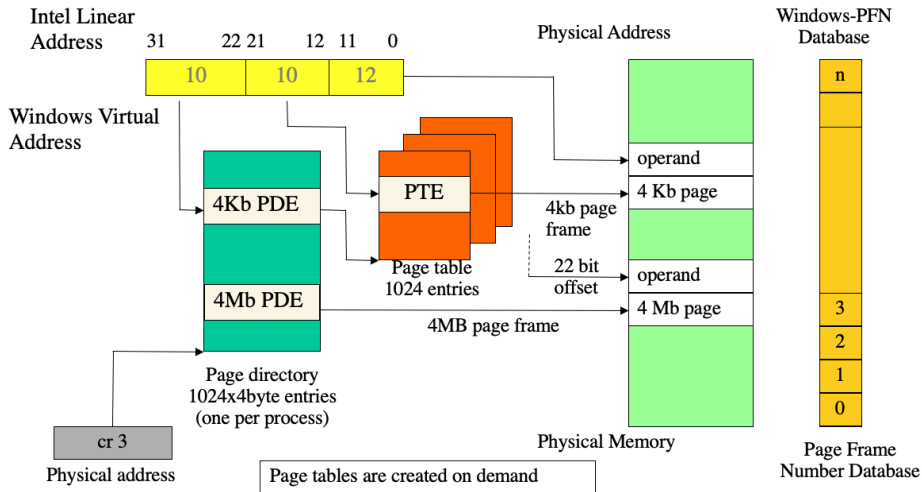
地址转换过程



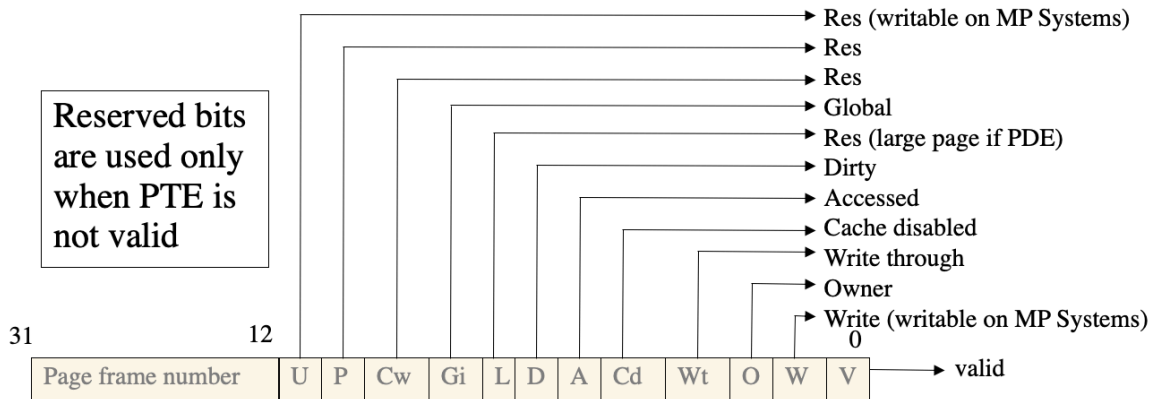
页表自映射机制



基于 4KB 页面的 X86-32 二级页表



X86-32 页表项结构

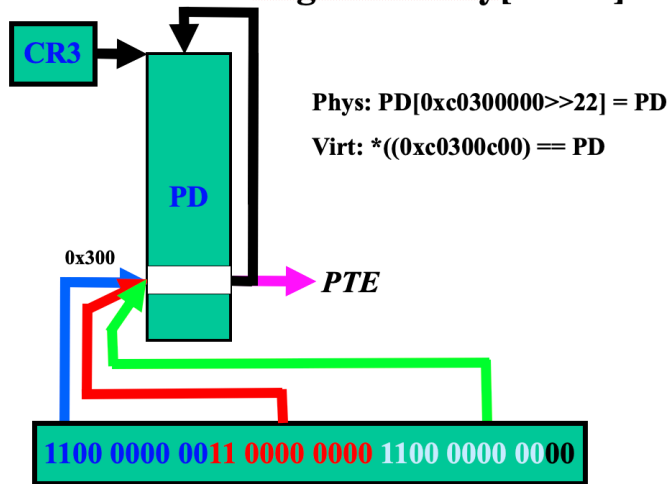


地址转换中的虚拟地址字段获取 (C 语言)

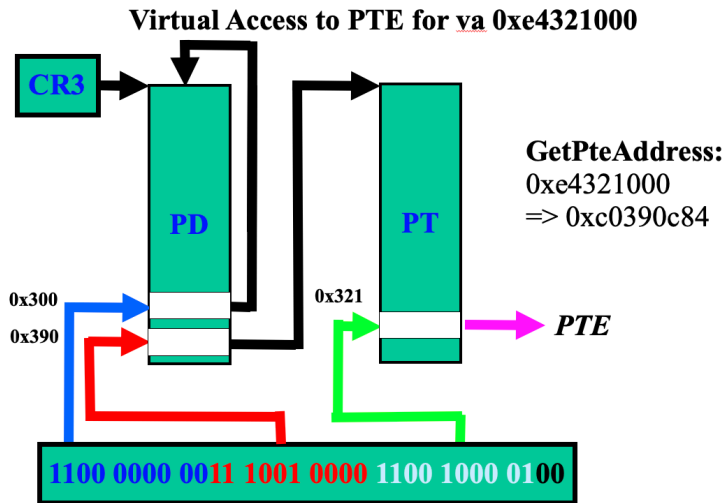
```
// page directory index
#define PDX(la) (((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
// page table index
#define PTX(la) (((uintptr_t)(la)) >> PTXSHIFT) & 0x3FF)
// page number field of address
#define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)
// offset in page
#define PGOFF(la) (((uintptr_t)(la)) & 0xFFF)
```


X86-32 的第一级页表自映射

Virtual Access to PageDirectory[0x300]



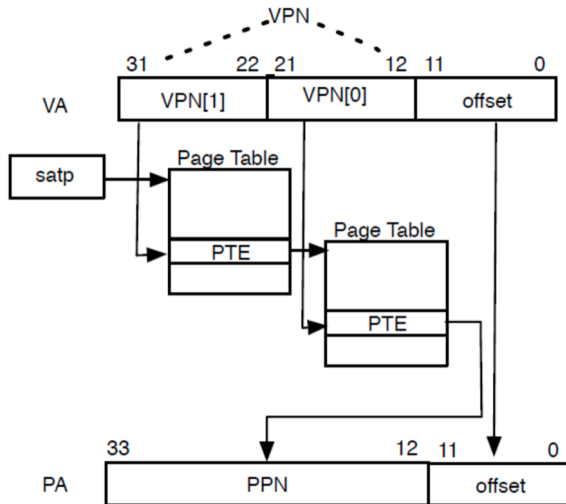
X86-32 的第二级页表的自映射



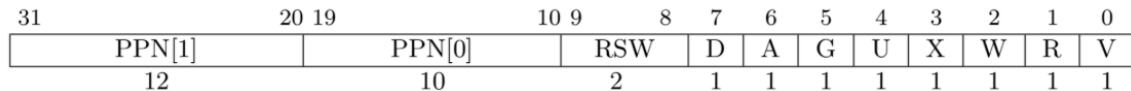
X86-32 自映射页表项初始化 (C 语言)

```
// recursively insert boot_pgdir in itself  
// to form a virtual page table at virtual address VPT  
boot_pgdir[PDX(VPT)] = PADDR(boot_pgdir) | PTE_P | PTE_W;
```

基于 4KB 页面的 RISC-V Sv32 二级页表



RISC-V32 页表项结构: Sv32 页表项格式

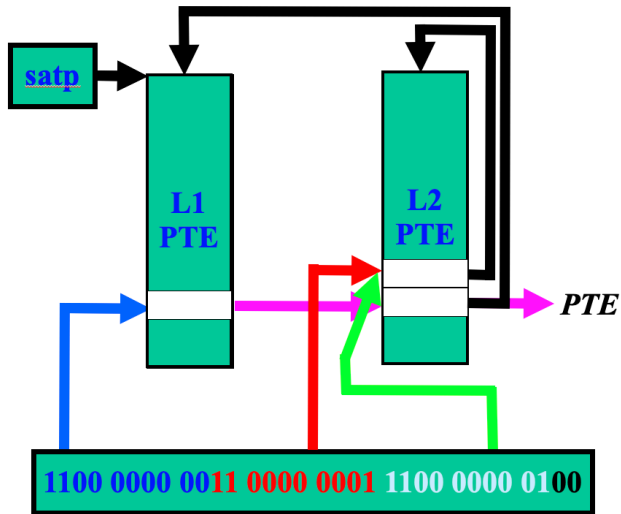


- 如果 X, W, R 位均为 0 , 则表示该项包含了下一级页表的物理地址 (为页目录项) 。
- 否则表示该项包含了页面的物理地址 (一般为页表项) 。

RISC-V32 页表项结构：页表项 R/W/X 字段含义

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	<i>Reserved for future use.</i>
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	<i>Reserved for future use.</i>
1	1	1	Read-write-execute page.

rCore 中 riscv-Sv32 自映射



- RISC-V 页表项中的 flags，明确表示它指向的是数据页 (VRW)，还是下层页表 (V)。
- 在访问一级页表虚地址期间，将它所对应的二级页表项 flags 置为 VRW。
- 访问二级页表本身，还需要再加一个自映射的二级页表项，其 flags 为 VRW。