

# Operating

## System - P46

最优页面置换算法 (optimal page replacement algorithm) 是一种理论上保证最少的缺页率的理想化算法

指在缺页中断发生时，在内存中有一个页面包含着紧接着的下一条指令的下一个页面

而每个页面都可以用在该页面首次被访问前所要执行的指令数作为标记

最优页面置换算法规定应该置换标记最大的页面

以此把因需要被淘汰的页面而发生的缺页中断推迟到将来

唯一的问题是 最优页面置换算法是无法实现的

即在缺页中断发生时，OS 无法知道各个页面下一次将在什么时候被访问

类似于在批处理系统中的最短作业优先调度算法 (shortest job first)

即系统如何知道哪个是最短作业

注意：可以通过首先在仿真程序上运行程序，并跟踪所有页面的访问情况

在之后的运行中利用仿真运行时收集的信息，是可以实现最优页面置换算法

但是特别注意，页面访问记录仅仅针对测试过的程序和特定的输入

因此据此推导出的 最优页面置换仅会对特定的程序和输入

所以这个方法在实际系统中却无法使用

最近未使用页面置换算法 (Not Recently Used)，为使 OS 能够收集有用信息，为每一个虚拟页面设置两个状态位

R 位：当页面被访问（读/写）时设置，M 位：当页面被写入（修改）时设置

由硬件设置两个状态位，当状态位设置为 1 后，则保持 1 直到 OS 将其复位

当进程启动时，进程所有页面的两个位都由 OS 设置为 0

R 位被定期地（比如每次时钟中断时）清零，以区别页面是否在最近被访问

M 位不会定期地清零，由于需要 M 位决定淘汰时是否写回磁盘

当缺页中断发生时，OS 检查所有页面并根据 R 位/M 位分类

第 0 类：没有被访问 (R)，没有被修改 (M)

第 1 类：没有被访问 (R)，已被修改 (M)

第 2 类：已被访问 (R)，没有被修改 (M)

第 3 类：已被访问 (R)，已被修改 (M)

NRU 算法随机地从类编号最小的非空类中选一个页面淘汰

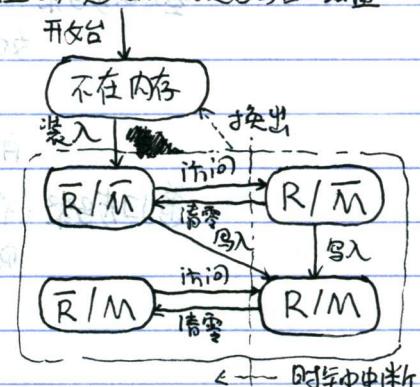
隐含的意思是，在最近的一个时钟中断中

没有被访问过的已修改的页面比被频繁使用的“干净”页面适合被淘汰

即第 1 类（由第 3 类清零 R 位得到）比第 2 类适合淘汰

优点是易于理解且能够有效地被实现

缺点是并非性能最好的算法，但通常句多用



# Operating

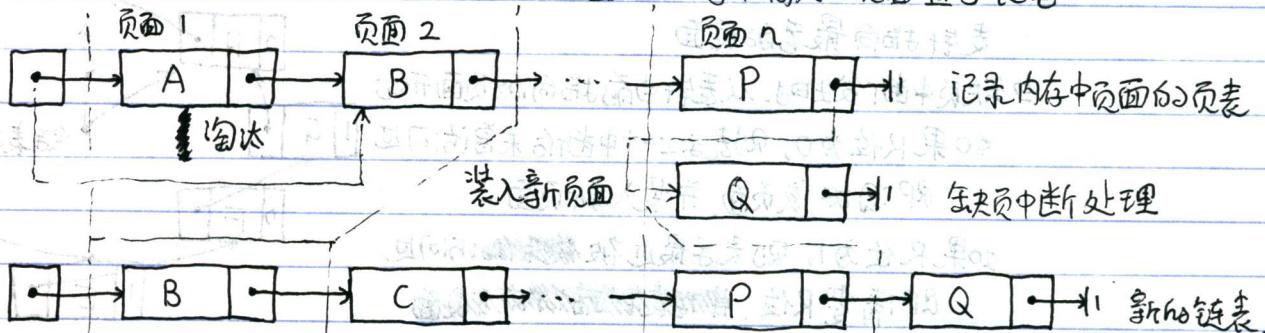
## System - P47

先进先出页面置换算法(First-In First-Out, FIFO), 是一种开销较少的页面置换算法。

即OS维护一个所有当前在内存中的页面的链表。

其中最新进入的页面放在表尾，最早进入的页面在表头。

当缺页中断发生时，淘汰表头的页面，并把新调入的页面置于表尾。



优点是算法易于实现，结构简单，如链表可用单链表也可用双链表。

缺点是算法可能淘汰常用页面，导致在短时间内产生新的缺页中断。

于是很少使用纯粹的FIFO算法，但有很多衍生的算法。

第二次机会页面置换算法(second chance)，用于避免FIFO算法经常淘汰常用页面的问题。

设置一个检查位R，用于记录页面是否被访问过。

当发生缺页中断时，检查链表头页面的R位。

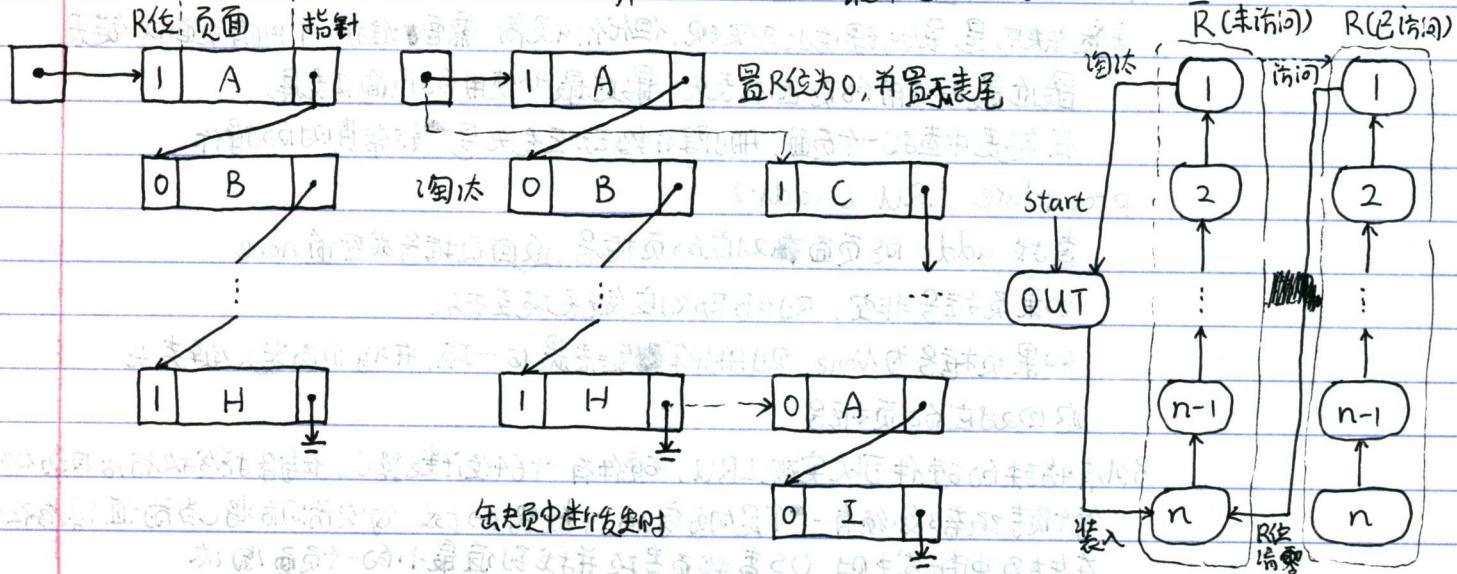
如果为1，则将页面移到表尾，并置R位为0，再检查链表的下一项。

如果为0，则表示该页面是最早进入且未被访问过的页面，即可以淘汰。

或者写回磁盘（已被修改），或者直接淘汰（“干净”的）。

如果所有页面的R位均为1，则会遍历一次所有页面并清空所有R位。

此时如纯粹的FIFO算法一样淘汰最早进入的页面。



# Operating System - P48

时钟页面置换算法(CLOCK), 与第二次机会算法类似, 但避免了第二次机会算法需要频繁移动链表项  
即如果一个链表项的R位为1, 则清零R位, 并将该表项移至链表尾部

CLOCK算法把页面保存在环形链表中

环形链表的页数即内存的实际页框数

表针指向最老的页面

当缺页中断发生时, 从表针当前指向的页面开始

如果R位为0, 则表示上一个中断后未曾访问过

即淘汰该页面, 并装入新页面

如果R位为1, 则表示最近被划去访问过

即清零R位, 划去

并将表针指向下一个页面, 可知此时指针指向的为最新的页面

#define M 8 实际内存页框数

#define point O 表项指针

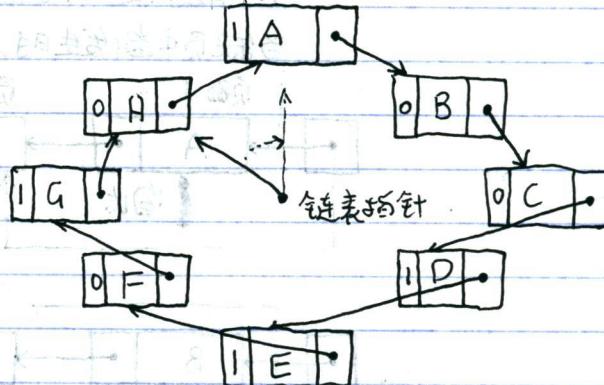
procedure CLOCK (装入页面A)

while  $\text{point} \neq \text{null}$

if  $\text{P.R} == 0$  then 淘汰P, 并装入A, 退出循环

else  $\text{P.R} = 0$ ,  $\text{point} + 1 \bmod M$

$\text{Point} + 1 \bmod M$



最近最少使用页面置换算法(Least Recently Used, LRU), 是对最优页面置换算法的近似

即基于观察: 在之前数条指令中频繁使用的页面很有可能在之后的数条指令中被使用

或者说, 已经长时间未被访问的页面很可能在未来较长时间内仍然不会被访问

即在缺页中断发生时, 置换未使用时间最长的页面

注意缺点是虽然理论上可实现, 但代价很高, 需要维护一个所有页面的链表

最近最多使用的页面在表头, 最近最少使用的页面在表尾

在链表中查找一个页面, 删掉并移动至表头是非常费时的操作

procedure LRU (addr)

查找 addr 的页面对应的页框号, 返回页框号或空值 none

如果页框号非空, 则移动对应链表项至开头

如果页框号为 None, 则删除链表最后一项, 并将页面装入链表头

返回对应的页框号

另外有特殊的硬件可以实现LRU, 硬件有一个64位计数器C, 在每条指令执行后自动加1

每个页表项都必须有一个足够容纳计数器的域, 每次访问后将C当前值保存在页表项中

在缺页中断发生时, OS查找页表项并找到值最小的一个页面淘汰

# Operating System - P49

SFSU

最不常用算法(Not Frequently Used, NFU) 为一个用软件实现 LRU 的可能解决方案

用于解决只有非常少的计算机拥有 LRU 所需硬件的问题

NFU 算法将每一个页面与一个软件计数器相关联，计数器初值为 0

每次时钟中断时，由 OS 扫描内存中的所有页面，将页面的 R 位 (0 或 1) 加到其计数器上

可知页面的计数器大致上跟随着该页面被访问的步数程度

当缺页中断发生时，可置换计数器值最小的页面

NFU 的主要问题是从不忘记任何事情。

如在一个多次(扫描)编译器中，在第一次扫描中频繁使用的页面在第二次扫描中，仍有高计数器值

如果第一次扫描的执行时间恰为最久的，则含有其后扫描代码的页面计数器均小于第一个页面

于是缺页中断发生时，OS 将置换有用页面而非不再使用的页面

NFU 可以修改为老化算法(Caging)，以很好地模拟 LRU，即在每个时钟中断时

首先，将计数器右移一位，如  $1101100 \rightarrow 01101100$

然后，将 R 位的值加到计数器的最左一位。如 R: 1  $\downarrow$  11101100

时钟滴答		(1)	(2)	(3)	(4)	(5)
R 位:	页面	101011	110010	110101	100010	011000
0	1000 0000	1100 0000	1110 0000	1111 0000	0111 1000	
1	0000 0000	1000 0000	1100 0000	0110 0000	1011 0000	
2	1000 0000	0100 0000	0010 0000	0001 0000	1000 1000	
3	0000 0000	0000 0000	1000 0000	0100 0000	0010 0000	
4	1000 0000	1100 0000	0110 0000	1011 0000	0101 1000	
5	1000 0000	0100 0000	1010 0000	0101 0000	0010 1000	

当缺页中断发生时，NFU 会置换计数器值最小的页面，如在时钟滴答 5 后，置换页面 3

或者按位比较，可知在时钟滴答 2 与 3 之间，页面 3 和 5 被访问之后再无访问

但是在时钟滴答 0 与 1 之间页面 5 被访问过，所以置换页面 3

NFU 老化算法的问题：

无法区分在一个时钟滴答中页面的访问顺序和频率，因为每个时钟滴答只记录一位

老化算法的计数器只有有限位数，于是限制了对以往页面的记录

即无法记录位数的时钟滴答之前的页面访问情况。

# Operating System - P50

请求调页 (demand paging), 指页面在需要时调入, 而不是预先装入的策略

注意在单纯的分页系统中, 刚启动进程时, 在内存中并没有页面

于是从CPU读第一条指令到访问全局数据和堆栈会频繁引起缺页中断

而在一段时间后进程才能在较少缺页中断的状态下运行, 此时所需页面大部分已在内存中

访问局部性 (locality of reference), 对于局部性原理 (principle of locality)

指取决于存储器访问模式的频繁访问相同值或相关存储位置的现象 memory access pattern  
the same values, or related storage locations, are frequently accessed, depending on the

两种基本类型: 时间局部性 (temporal locality) 和 空间局部性 (spatial locality)

时间局部性: 指在相对较小的持续时间内对特定数据 和/或资源的重用

the reuse of specific data, and/or resources, within a relatively small time duration

空间局部性, 指在相对靠近的存储位置内使用数据元素

the use of data elements within relatively close storage locations

顺序局部性 (sequential locality), 为一种空间局部性的特殊情况

指当数据元素 线性地排列和访问时, 如遍历一维数组的元素  
occurs when data elements are arranged and accessed linearly  
such as traversing the elements in one-dimensional array

另外还有其他几种不同类型的访问局部性

内存局部性 (memory locality), 指显式地与存储器相关的空间局部性

spatial locality explicitly relating to memory

分支局部性 (branch locality)

指在空间-时间坐标空间中路径的预期部分只有少数可能的替代方案

only a few possible alternatives for the prospective part of

the path in the spatial-temporal coordinate space

当指令循环具有简单结构, 或者小的条件分支指令系统的可能结果限制在小部分可能情况

注意分支局部性通常不是空间局部性, 因为分支可能彼此远离

等距局部性 (equidistant locality), 介于空间局部性和分支局部性之间

考虑以等距模式访问位置的循环, 简单的线性函数可以预测将来访问的位置

a loop accessing locations in an equidistant pattern

in the near future

a simple linear function can predict which location will be accessed

为了受益于频繁出现的时间局部性和空间局部性, 大多数信息存储器是分级的 (hierarchical)

对于分支局部性的情形, 当代处理器具有复杂的分支预测器 (sophisticated branch predictors)

# Operating System - P51

工作集

(working set), 指一个进程当前运行正在使用的页面的集合

如果整个工作集都装入内存，则进程在运行至下一阶段（如编译器下一遍扫描）前，不会产生很多缺页中断  
而如果内存过小而无法容纳整个工作集，则进程运行过程中产生大量的缺页中断

颠簸 (thrashing), 又称抖动，指在程序运行过程中，每执行数条指令程序就发生一次缺页中断

通常是在淘汰了一个页面后，被淘汰的页面是一个很快会被再次访问的页面

则在上一次缺页中断后，又很快发生新的缺页中断

由于执行一条指令需要几个纳秒，而从磁盘读取页面需要约十毫秒

所以颠簸会导致程序运行速度变得非常缓慢

通常的解决策略有

如果由页面替换策略引起，则修改替换算法

如果由运行程序太多引起，即无法将不同程序的频繁访问页面调入内存，则降低多道程序数量

否则或者终止进程，或者增加物理内存容量

工作集模型 (working set model)，目的在于大大减少缺页中断率

预先调页 (prepaging)，指在进程运行前预先装入其工作集页面

为此分页系统需要跟踪进程的工作集，以确保在进程运行以前，其工作集已在内存中了

$w(k, t)$  记为一个工作集，即在任一时刻  $t$ ，都存在一个集合，包含所有在最近  $k$  次内存访问过的页面

由于  $k > 1$  次访问必定访问了  $k-1$  次访问所访问的页面

所以  $w(k, t)$  是  $k$  的单调非递减函数

但是随着  $k$  的增大， $w(k, t)$  不可能无限增大

因为程序不可能访问其地址空间无法容纳的页面

这是一个渐近的过程，即存在一个很大的  $K$  值范围，处在这个范围内时工作集  $w(k, t)$  不会变

所以当程序重新开始时，可能根据其上次结束时的工作集对需要用到的页面作一个合理的推测

为了当缺页中断发生时，淘汰一个不在工作集中的页面，

需要一种精确的方法来确定工作集中的页面的方法 来实现页面淘汰算法

注意对于预先确定的  $K$  值，每次内存访问之后，其最近  $K$  次内存访问所使用的页面集合就是唯一确定的

一种方法是含有才  $K$  长度  $k$  的多位寄存器，每次访问后多位寄存器左移一位，并将当前页面号存入最末位

但是缺页中断读出内容并排序并删除重复页面，以获取工作集，但其开销巨大

另一种方法是，定义工作集为过去一段时间（如 10ms）中内存访问所用到的页面集合

每个进程只计算其自身的执行时间，从其执行开始到当前所实际使用 CPU 时间称为当前实际运行时间

则进程的工作集可以定义为过去 2 秒实际运行时间中所访问的页面的集合

五、缺页中断

淘汰策略

最近最少使用

最近最常使用

先进先出

随机淘汰

最近未使用

# Operating

## System - P52

基于工作集的页面置换算法，即找出一个不在工作集中的页面，并淘汰之。

由于只有在内存中的页面才可以被淘汰，所以算法忽略不在内存中的页面。

R位：上次使用时间  
M位：其他算法不需要的页表项，如页框号，保护位，M位

页表	1   2084	0   121308	1   2003	0   1620	...
----	----------	------------	----------	----------	-----

假定在每个时钟中断发生时，或者用硬件或软件方法来设置和清除R位。

对于给定的时间C，定义工作集为过去的C秒实际运行时间中访问过的页面集合。

当缺页中断发生时，记录当前时间T，用于计算页面至上次使用时间的间隔，称为生存时间。

算法顺序地扫描所有页面，根据时间间隔和R位来选择淘汰页面。

如果  $R=0$  且 生存时间  $> C$  ] 即页面不在工作集中，且在最近的一个时钟中断中未被访问。

则淘汰该页面。

如果  $R=0$  且 生存时间  $\leq C$  ] 页面在工作集中，但是在最近的一个时钟中断中未被访问。

则记住此类页面中 上次使用时间 最小的一个。

如果  $R=1$  ] 如果所有页面都访问过，则随机选取，或选择未修改的页面。

则更新该页面的上次使用时间为当前时间。

如果在扫描过程中没有淘汰过页面，则选择第二类中最久未被访问的页面淘汰。

否则在页面中选择一个淘汰，可以使用其他页面置换算法。

工作集时钟页面置换算法 (WSClock Page Replacement Algorithm)。实现简单，性能较好的结合时钟算法的工作集算法。

主要使用的是上次使用时间，R位，M位。

上次使用时间

当缺页中断发生时，

指针顺序地检查每个页面。

1620	R位 M位	2032
0   1	1   0	1   1

如果指向页面  $R=0$ ，且 生存时间  $> C$  ]

如果  $M=0$ ，则置换该页面。

2003
1   0

2020
1   1

如果  $M=1$ ，则页面修改过。



提交申请将此页面写回磁盘。

1980
1   1

2014
1   0

如果指向页面  $R=1$ ，则设置  $R=0$ 。

1213
0   0

1010
0   0

然后指针指向下一个页面。

由于所有的页面都可能在某个时钟周期调度磁盘I/O，为了降低磁盘阻塞，可设定最大只允许写回n个页面。

在指针转完第一圈后如果没有淘汰页面，则第二圈存在两种可能。

至少调度了1次磁盘写操作，则至少有一个  $R=0$  且 生存时间  $> C$  的页面写回磁盘向 M位设置为 1。

没有调度过磁盘写操作，则所有的页面都在上一个时钟中断访问过，即  $R=1$ 。

且在上一圈中所有页面 R 都清除为 0，则可能存在  $R=0$  且 生存时间  $> C$  的页面。