文章已收录我的仓库：Java学习笔记与免费书籍分享

页面置换(淘汰)算法

> 本文基于《现代操作系统》总结

前言

对于操作系统而言，虚拟空间是非常大的，我们往往无法直接将如此大的空间装入内存，而即使我们采用多级页表与段页式存储即使，也仅仅只是节省了页表的大小，如此将如何多的物理页装进内存仍然是一个问题，为此科学家们提出了一种动态的思想，当剩余空间不够装入新页面时，操作系统就选择适当的页将其刷新回磁盘(如果确实是脏页的话，如果不是脏页则仅仅只是覆盖)，以此空出空间装入新页，而这则需要运用到页面置换算法。

还有一个问题是：是否需要对每一个进程固定一个空间大小，换句话说，是采用**局部分配**策略还是**全局分配**策略。

采用局部分配的好处在于每一个进程空间都是互不影响的，即使一个进程频繁的发生页面置换也不会影响到其他进程，并且易于管理；但缺点是，如何分配一个进程的空间大小是难以确定的，有的进程可能频繁发生页面置换，而有的进程可能只会使用一点点空间，这是预先无法确定的，如果给一个进程分配的空间小了，那么该进程可能会发生**内存“颠簸”**(频繁的发生页面置换)。

若采用全局分配，一个进程理论上可以使用整个内存空间，内存大意味着发生置换的频率就小了，在绝大多数情况下，全局分配优于局部策略。但缺点也是很明显的，如果进程较多，一个进程频繁进行页面置换可能会置换其他进程的页，这可能会导致其他进程也造成

缺页异常，需要引入新页，从而造成死循环。这是操作系统不愿意看到的事情，而且操作系统难以控制这种局面。

好消息是历经这么多年，现代操作系统已经很聪明了，现代操作系统大多采用局部分配的方法，这更易于操作系统控制。当操作系统开始分配页面时，会根据进程数量、进程资源大小等参数相对公平的分配固定的空间，例如总内存大小为 10kb，当已有一个资源为 5kb 的进程A正在运行，由于只有一个进程，进程A的空间大小为整个内存大小 10kb，现在操作系统打算为资源大小为 15kb 的进程B分配空间，操作系统根据进程大小与进程数量进行评估，由于进程B大小：进程A大小 = 3：1，因此操作系统为B分配 3/4 的内存空间大小 7.5kb，同时调整A的大小为 2.5kb(淘汰页面)。

而在运行时，操作系统仍然会聪明的进行动态调整，如果一个进程A频繁的发生颠簸，而进程B几乎不发生颠簸，操作系统会适当增加进程A空间的大小，而减少进程B空间的大小。如果所有的进程都发生颠簸，这是最坏的情况，此时操作系统会选择一些进程将它们暂时的调换到磁盘中以空处一些空间，直到有新的空间时再将他们换回来。

何时会发生页面置换？**当发生缺页异常时(页表项中存在位为0)，需要引入该页，如果此时进程的内存空间不够，则需要淘汰一些页面。**对于某些操作系统而言，可能会预先载入某些页以减少缺页异常的发生。

在了解这些之后，让我们来具体探讨一些页面淘汰算法。

最优置换算法(OPT)

这个算法是不可能实现的，但面试需要说出这个算法。

最优置换算法的思想就是选取**最晚使用的页面**淘汰，这当然是最优的算法，但问题是计算机根本也不知道各个页面下一次将在什么时候被访问，因此该算法是无法实现的。

该算法通常作为基准，其他算法与该算法进行对比。

先进先出页面置换算法(First-In First-Out，FIFO)

顾名思义，淘汰最先使用的页面，很容易想到队列这种数据结构，从尾入，从头出。

具体的实现是由操作系统维护一个所有当前在内存中的页面的链表，每一次最新进入的页面放在表尾，那么表头页面即使最久使用的。当发生缺页中断时，淘汰表头的页面并把新调入的页面加到表尾。

FIFO的实现跟简单，但效率不是很高。FIFO的算法思想很纯粹，即淘汰使用最久的页面，但是，保不准某些页面使用的久而且使用的频繁，这样淘汰最久使用的页面就不是一个好的办法了，因为下一次很可能又会载入该页面，因此很少有操作系统采用存粹的FIFO算法了。

第二次机会页面置换算法(Second Chance，SC)

SC是对FIFO的一个优化，在FIFO中可能会错误的淘汰掉接下来可能会使用的页面，因此SC给予这些页面第二次机会。

SC仍然维护一个FIFO链表，但同时对每一个页会维护一个R位，为0则表示该页未使用，为1表示该页最近使用过，每次访问页都会设置该页R位为1。

当触发缺页异常，SC的算法规则是：

- 从表头移除页，如果该页R=1，则表示该页最近被使用过，可能会接着被使用，给予其第二次机会，因此操作系统将该页重新装入表尾，但设置R=0，然后继续从表头读页。
- 如果R=0，则表示该页生存时间很久了，并且最近都没有使用，则可以像FIFO那样直接置换掉该页。

SC的思想是NRU思想，我们很快就会介绍NRU算法思想，实践表明SC算法是一个较为合理的算法，但缺点是SC经常要在链表中移动页面，这是非常耗时的。

最近未使用页面置换算法(Not Recently Used，NRU)

NRU选择淘汰最近未使用的页面，NRU算法认为如果最近使用了一个页面，那么很有可能会继续使用，而对于没有使用的页面，可能很久都不会再使用了，因此淘汰掉该页面。

在大部分具有虚拟内存的计算机 中，在页面的页表项中，系统为每一页面设置了两个状态位。当页面被访问（读或写）时设置R位； 当页面（即修改页面）被写入时设置M位。R位是NRU实现的关键，R位为 1 表示该页最近被访问，为 0 则表示未被访问。

由于页面的访问是相当频繁的，因此设置R、M位最好是由硬件来完成，好在这项工作不算复杂，市面上已经有类似的实现了，

问题的关键是，这个最近的含义是什么？通常情况下，NRU采取操作系统的时钟中断(通常20ms一次)作为标志，即记录从上一次时钟中断到目前为止的访问信息，这意味着每一次时钟中断需要设置R为0，以表示“最近周期”的重新开始。时钟中断一般不设置M位，M位是脏位的标志，当确实要置换某页时，如果该页M位为1，表示页被修改，则必须要写回磁盘以维护一致性；如果M位为0，则页未被修改，可以直接覆盖(磁盘中的页与内存中的页一致)，无须写回。

当发生缺页中断时，操作系统检查所有的页面并根据它们当前的R位和M位的值，把它们分为4类：

- 编号0：没有被访问，没有被修改(最近未使用的干净页)。
- 编号1：没有被访问，已被修改(最近未使用的脏页)。
- 编号2：已被访问，没有被修改(最近使用的干净页)。
- 编号3：已被访问，已被修改(最近使用的脏页)。

NRU算法随机地从编号最小页面中挑选一个淘汰，选择的思想是优先选择干净页，这样可以节省一次写回磁盘的时间。

NRU主要优点是易于理解和能够有效地被实现，但它的性能不是很好，因为他必须要扫描所有表项才能找出要淘汰的页。

时钟算法

时钟算法属于NRU软件实现的一种方法，算法思想源自于上述标准的NRU算法，上述NRU算法采用操作系统时钟中断周期作为标志，而时钟算法维护一个循环链表，以一个循环为标志。

时钟算法仍然维护一个R位标识是否被访问，1 标识被访问，0 标识未被访问，只不过这一次由软件实现，链表被放置在内存中(当然只会放置指向页或页表项的指针)。

具体的算法是：

- 访问一次页面，则设置该页R位为1
- 当发生缺页异常时，如果此时进程空间不够，需要淘汰页面，则：
    - 检查当前页R位，如果是 0，表示该页最近未使用，则置换该页，检查页表项判断是否需要刷新回磁盘。

        > 一种权衡是：如果该页是脏页，是否需要选择置换该页？答案是不确定的，选择置换该页则需要付出一次磁盘IO的代价；而不置换该页则需要继续查找，这也需要一定的时间，最坏的情况是所有页都是脏页，此时得等待操作系统发生时钟中断将脏页刷新回磁盘。

    - 如果为 1，则置为 0，并前移指针。

最坏情况下，时钟算法需要遍历一个周期才能找到R=0的页，这是非常糟糕的。

算法的思想是：如果当前指针指向的页为R = 0，则可以确定至少一个循环内该页未被访问，这是由于既然该页R位为0，要么该页从来没有被访问；要么该页在上一个循环内被设置为0，而一个循环内未被访问(如果访问了就是1了)；无论怎样，都至少经历了一个循环，则可以近似的将其认为最近使用的页面。

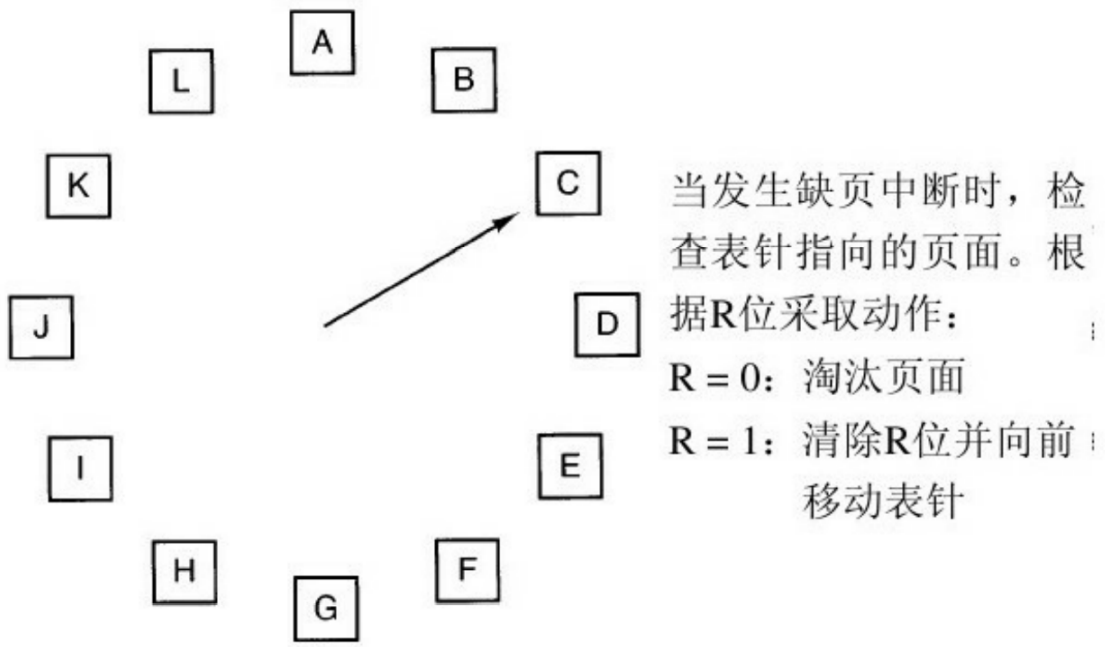


图　3-16　时钟页面置换算法

时钟算法是NRU软件实现的一种较好的办法，实现简单，效率也还行，通常是一种不错的实现方法。

最近最少使用页面置换算法(Least Recently Used，LRU)

LRU与NRU的区别在于：**NRU是严格查找最近未使用的页面，当最近页面都使用过，NRU就会变得不知所措，此时NRU通常会随机选取一个页面，这是造成NRU效率低效的主要原因；而LRU充分考虑到了这一点，即使最近页面都使用过，LRU也能找到最少使用的那么页面，实施淘汰算法。**从集合角度说，NRU包含于LRU，NRU通常作为LRU的近似实现。

因此LRU是最十分高效的算法，虽然LRU在理论上是可以实现的，但代价非常高。为了完全实现LRU，需要在内存中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。每次访问一个页时，找到该页在链表中的位置(可以利用哈希)，然后将它移动到表头，淘汰时选取表尾的页淘汰即可。

算法思想是很简单的：每次被访问页面都被移动到头，那么尾部的节点就是最近最少被访问的页了。换句话说，对于任意一个节点而言，在它前面的节点一定是比它后被访问的，那么该节点在这段时间内访问的频率不如在它前面的节点，当然，这还要看对“最近”是如何定义的，对于这里的LRU而言，“最近”被定义为上一次访问尾节点的时间到现在。

但是，页面的访问是非常频繁的，诸如`int a = 0; a++;`这样的语句都需要访问对应的页，毫不夸张的说，几乎99%代码语句可能都需要访问页，而每次访问又需要移动页面指针，这代价太高了，因此这种存粹的**软件模拟**在操作系统层面上来说是不可取的。必须要借助硬件支持。

有一种很巧妙的方法实现LRU，即：每个页表项中存在一个时间字段，当访问该页时，设置该时间字段为当前时间，淘汰时，扫描全部页表项，找到最晚的时间戳淘汰它。

我们必须要利用硬件加速这个过程，因为软件更新时间戳效率太低了，这种算法是想要精确实现LRU的另一个选择。缺点是该算法与NRU类似，也必须扫描全部页表项，代价也是较高的(但可以接受)，并且还要考虑时间戳溢出问题。

最不常用页面置换算法(Not Frequently Used，NFU)

该算法将每个页面与一个软件计数器相关联，计数器的初值为0。每次时钟中断时，由操作系统扫描内存中所有的 页面，将每个页面的R位（它的值是0或1）加到它的计数器上，然后重新设置R位为0。这个计数器大体上跟踪了各个页面被访问的频繁程度。发生缺页中断时，则置换计数器值最小的页面。

NFU与LRU的区别就在于NFU淡化了“最近”的含义，是一种近似LRU的实现，NFU从不忘记任何事，他记得所有历史以来所有事情。这就会产生一些问题，比如，在一个多次（扫描）编译器中，在前几次扫描中被频繁使用的页面在程序进入第N次扫描时，其计数器的值可能仍然很高(甚至可能会发生溢出)，尽管该页可能前面使用频繁而后面使用不频繁。

因此，我们得想办法让NFU忘掉一些事情。

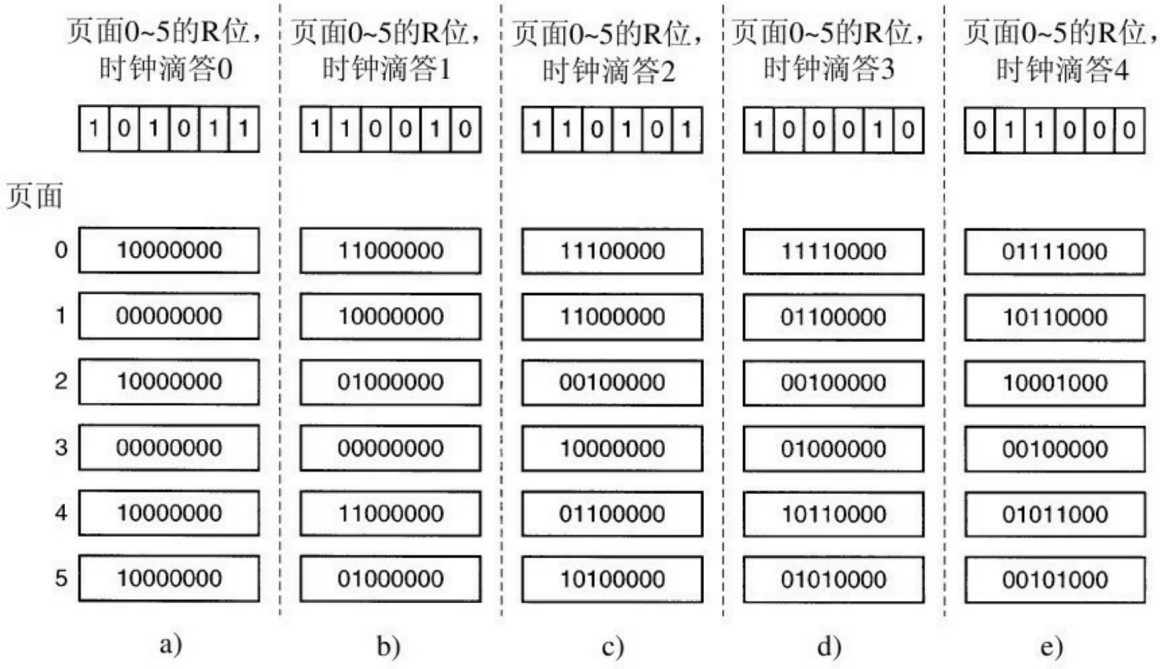
老化算法

改进后的算法叫做老化算法，只需对NFU做一个小小的修改就能使它很好地模拟LRU。其修改分为两部分：

- 首先，在R位被加进之前先将计数器右移一位(对所有的计数器执行)；
- 其次，将R位加到计数器最左端的位上。

改进的思想是：右移一位使得NFU能够忘掉最近没被用过的页面(右移相当于除以2，数减小)，而由于所有计数器被右移一位，衰减是非常快的(加在左边一下就衰减没了)，因此相加应该加在右边的位上(恰好右移一位空出)。

考虑经历了5个时钟周期，老化算法的运行如下所示：



当发生缺页时，置换计数器最小的值，例如在滴答4时，应该置换掉页面3，页面3在滴答2出现了一次，是所有页中最晚使用的页(页0,1,2,4在嘀嗒3,4使用，页5虽然在嘀嗒2使用，但页5已经是第二次被使用了)。

LRU和老化算法的区别是老化算法的计数器只有有限位数（本例中是 8位），这就限制了其对以往页面的记录。

工作集页面置换算法

人们很早就发现大多数程序都不是均匀地访问它们的地址空间的，而访问往往是集中于一小部分页面。一次内存访问可能会取出一条指令，也可能会取数据，或者是存储数据。在**任一时刻t，都存在一个集合，它包含所有最近k次内存访问所访问过的页面**。这个集合w(k,t)就是工作集(k是人为确定的)。

工作集与 LRU 的区别是，工作集并不考虑频度的问题，它只在乎最近该页在不在工作集内，而不考虑该页使用频率如何，换句话说，被淘汰的页面不在工作集内，但并不能保证该页面是最近最少使用的。

工作集页面置换算法思想是当某一时刻 t 需要置换页面时，选取不在 w(k,t) 内页面淘汰它。为实时维护工作集，我们需要在内存中维护长度为 k 的链表，每次访问页面时都将页号从尾放入链表中，如果链表已满，则需要从头弹出一个元素，但问题是每次访问页都需要调整链表，代价也十分高昂。

通常采取近似的方法，例如基于时间戳的方法，比如：工作集是过去 T ms 中的内存访问所用到的页面的集合。我们并不需要每次访问都更新时间戳，可以近似的只在时钟中断时进行更新，代价是时钟中断时不得不遍历整个表项进行更新。

现在扫描整个表项，会出现4种情况：

1. 时间戳不在工作集内，R=0：这种情况下直接进行置换即可，循环继续，但不再置换页面。
2. 时间戳不在工作集内，R=1：该页最近被访问，在等待中断更新，此时更新该页时间戳为当前时间，设置 R = 0。
3. 时间戳在工作集内，R=0：如果已经置换了页面，则忽略；否则，记录下该时间戳，如果所有都是这种情况，则选择时间戳最小的哪个淘汰。
4. 时间戳在工作集内，R=1：该页最近被访问，在等待中断更新，此时更新该页时间戳为当前时间， 设置 R = 0。

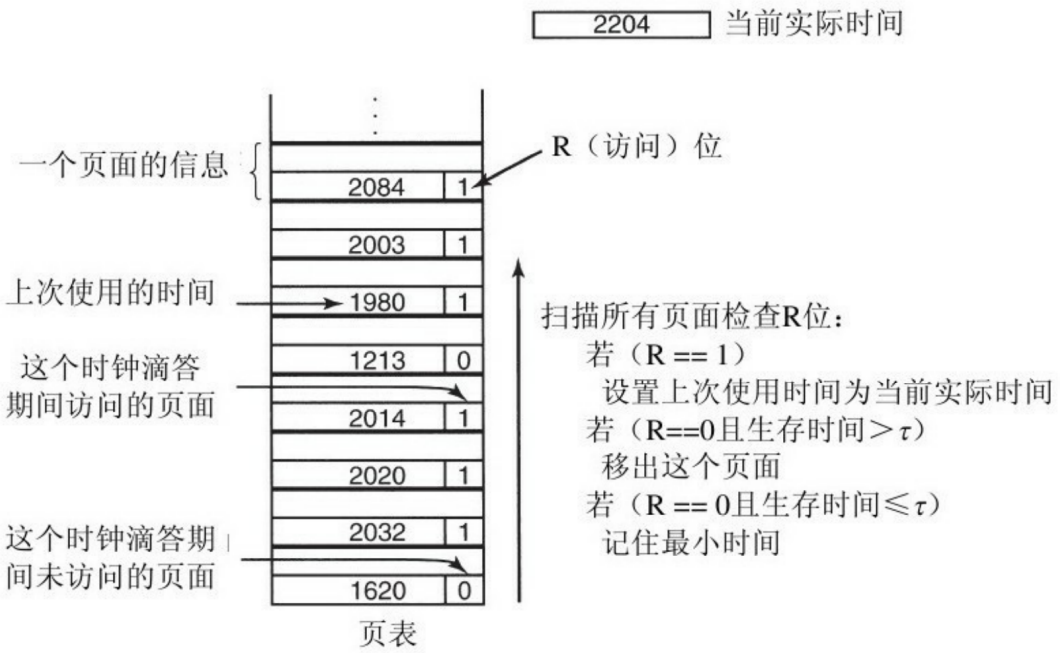


图　3-20　工作集算法

总结

算法	总结
OPT	不可能实现，只是作为比较的基准。
FIFO	实现简单，但可能换出重要的页，工作效果不好
SC	FIFO的优化，工作效果较好，但需要交换链表项，性能不是很高，但可以接受，是个不错的选择
NRU	LRU的近似，但工作效果不是很好，需要遍历全部页，性能不高
时钟算法	NRU的思想，LRU的近似实现，工作效果较好，性能较高，是个不错的选择
LRU(链表)	工作效果好，但每次访问都需要复杂的指针移动，性能低，非最优选

算法	总结
LRU(时间戳)	可取的LRU实现，但需要硬件更新时间戳，每个页都需要维护一个时间戳，空间成本高，而且**时钟是不可靠**的！
NFU	LRU的近似，工作效果不好，不可取
老化算法	NFU的优化，LRU的近似，工作效果好，但中断时需要遍历全部页表项，性能不高，但由于其工作效果优秀，是个不错的选择
工作集	工作效果一般，中断时需要遍历全部页表项，性能不高，非最优选