6.3 页面置换

1. 页面置换的发生时机

如果增加多道程序运行，就会过度分配(over-allocating)内存。例如运行 6 个进程，且每个进程有 10 页大小但事实上只使用其中的 5 页，那么就获得了更高的 CPU 利用率和吞吐量，且有 10 帧可作备用。然而，每个进程有可能会突然试图使用其所有的 10 页，从而产生共 60 帧的总需求，而实际物理内存只有 40 帧可用。

再者，还需要考虑到内存不是仅用于保存程序的页面。I/O 缓存也需使用大量的内存。这种使用会增加内存分配算法的压力。在 I/O 缓存和程序页面之间平衡内存的分配是个棘手问题。有的系统为 I/O 缓存分配了一定比例的内存，而另一些系统则允许用户进程和 I/O 与系统竞争使用所有系统内存。

内存的过度分配会导致以下问题：当一个用户进程执行时，发生一个缺页中断。此时操作系统会确定所需页在磁盘上的位置，但是却发现空闲帧列表上并没有空闲帧，所有内存都正在使用(见图 6.8) 。

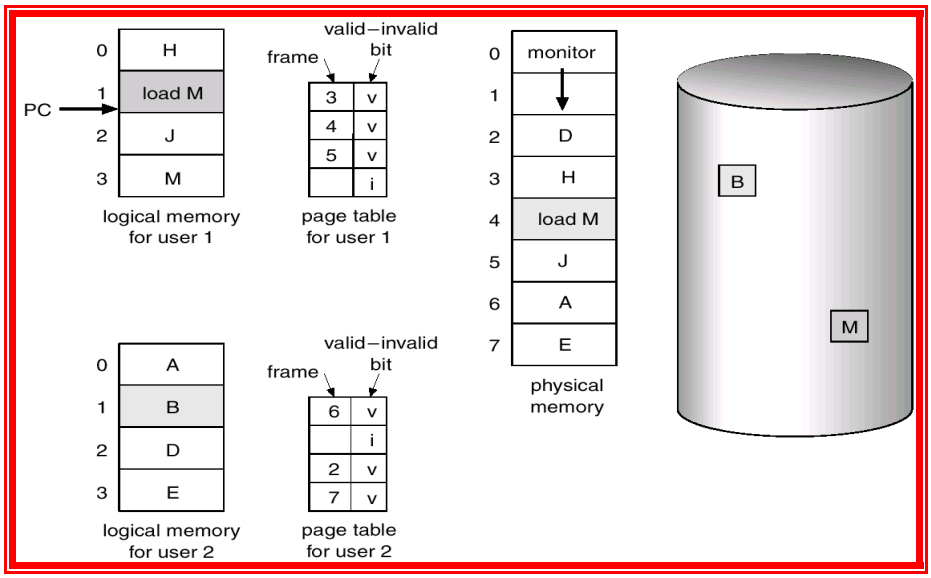

图 6.8 需要页置换的情况

这时操作系统会有若干选择。它可以终止用户进程。然而，按需调页是操作系统试图改善计算机系统的使用率和吞吐量的技术。用户并不关心其进程是否运行在调页系统上：调页对用户而言应是透明的。因此，这种选项并不是最佳选择。

操作系统也可以交换出一个进程，以释放其所有帧，并降低多道程序的级别。这种选择在有些环境下是好的，但更为常用的解决方法是页面置换(page replacement)。

在进程运行过程中，如果发生缺页，而此时内存中又无空闲块时，为了保证进程能正常运行，需要从内存中调出一页程序或数据送到磁盘的对换区。这个操作叫做页面置换。

决定将哪个页面调出必须根据页面置换算法(replacement policy)来确定。目标一般是将那些以后不再被访问或在较长时间内不会再被访问的页调出。置换算法通常会在局部性原理指导下依据历史统计数据进行预测。

2. 基本页置换

页置换采用如下方法。如果没有空闲帧，那么就查找当前未使用的帧，并将其释放。可采用这样的方式来释放一个帧：将其内容写到交换空间，并改变页表(和所有其他表)以表示该页不在内存中(见图 6.9) 。接着可使用该空闲帧来保存进程的缺页。同时还需在缺页处理

程序中添加页置换操作：

(1) 查找所需页在磁盘上的位置。

(2) 查找一个空闲帧:

a. 如果有空闲帧，那么就使用它。

b. 如果没有空闲帧，那么就使用页置换算法以选择一个"牺牲"帧(victim frame) 。

c. 将"牺牲"帧的内容写到磁盘上，改变页表和帧表。

(3) 将所需页读入(新)空闲帧，改变页表和帧表。

(4) 重启用户进程。

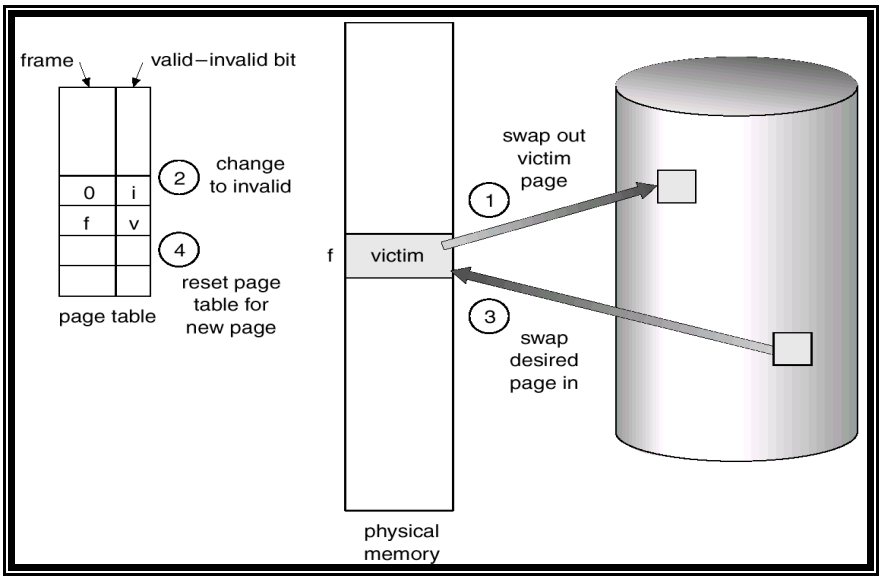

图 6.9 页置换

注意，如果没有帧空闲，那么需要进行两次页传输(一个换出，一个换入)。这种情况实际上把缺页处理时间加倍了，且也相应增加了有效访问时间。

上述过程可以通过使用修改位(modify bit)或脏位(dirty bit)以降低部分页置换的开销。每页或帧可以有一个修改位，通过硬件与之相关联。每当页内的任何字或字节被写入时，硬件就会设置该页的修改位以表示该页已修改。如果修改位已设置，那么就可以知道自从磁盘读入后该页已发生了修改。在这种情况下，如果该页被选择为替换页，就必须要把该页写到磁盘上去。然而，如果修改位没有设置，那么也就知道自从磁盘读入后该页并没有发生修改。因此，磁盘上页的副本的内容没有必要(例如用其他页)重写，因此就避免了将内存页写回磁盘上：它已经在那里了。这种技术也适用于只读页(例如，二进制代码的页)。这种页不能被修改。因此，如需要，可以放弃这些页。这种方案可显著地降低用于处理缺页所需要的时间，因为如果页没有修改它，可以降低一半的I/O时间。

页置换是按需调页的基础。它分开了逻辑内存与物理内存。采用这种机制，小的物理内存能为程序员提供巨大的虚拟内存。对于非按需调页，用户地址被映射到物理地址，所以这两地址可以不同。然而，所有进程的页必须在物理内存中。对于按需调页，逻辑地址空间的大小不再受物理内存所限制。例如一个具有 20 页的用户进程，可通过按需调页先用 10 个帧来执行它，如果有必要可以用置换算法来查找空闲帧。如果已修改的页需要被置换，那么其内容会复制到磁盘上。后来对该页的引用会产生缺页。这时，该页可以再调回内存，这时有可能会置换进程的其他页。

按需调页需要实现两个重要的算法，即帧分配算法(frame-allocation algorithm) 和页置换算法(page-replacement algorithm) 。如果在内存中有多个进程，那么必须决定为每个进程

各分配多少帧。而且，当需要页置换时，必须选择要置换的帧。因为磁盘 I/O 非常费时，所以设计合适的算法对系统性能起到至关重要的作用。即使请求页面调度方面的很小改进也会对系统性能产生显著的改善。

现有的系统有许多不同的页置换算法。每个操作系统可能都有其自己的置换算法。如何选择一个置换算法呢？通常需要一个最小的缺页率的算法。

可以这样来评估一个算法：针对特定内存引用序列，运行某个置换算法，并计算出缺页的数量。内存的引用序列称为引用串(reference string) 。可以人工地生成引用串(例如，通过随机数生成器)，或可跟踪一个给定系统并记录每个内存引用的地址。后一方法会产生大量数据(以每秒数百万个地址的速度)。为了降低数据量，可利用以下两个事实。

第一，对给定页大小(页大小通常由硬件或系统来决定)，只需要考虑其页码，而不需要完整地址。第二，如果有一个对页 p 的引用，那么任何紧跟着的对页 p 的引用决不会产生缺页。页 p 在第一次引用时已在内存中，任何紧跟着的引用不会出错。

例如，如果跟踪一个特定进程，那么可记录如下地址顺序：

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0611, 0120

如果页大小为 100B，那么就得到如下引用串：

1, 4, 1, 6, 1, 6, 1

针对某一特定引用串和页置换算法，为了确定缺页的数量，还需要知道可用帧的数量。显然，随着可用帧数量的增加，缺页的数量会相应地减少。例如，对于上面的引用串，如果有 3 个或更多的帧，那么只有 3 个缺页，对每个页的首次引用会产生一个缺页。另一方面，如果只有一个可用帧，那么每个引用都要产生置换，共产生 11 个缺页。通常，随着帧数量的增加，缺页数量会降低至最小值，如图 6.10。当然，增加物理内存就会增加帧的数量。

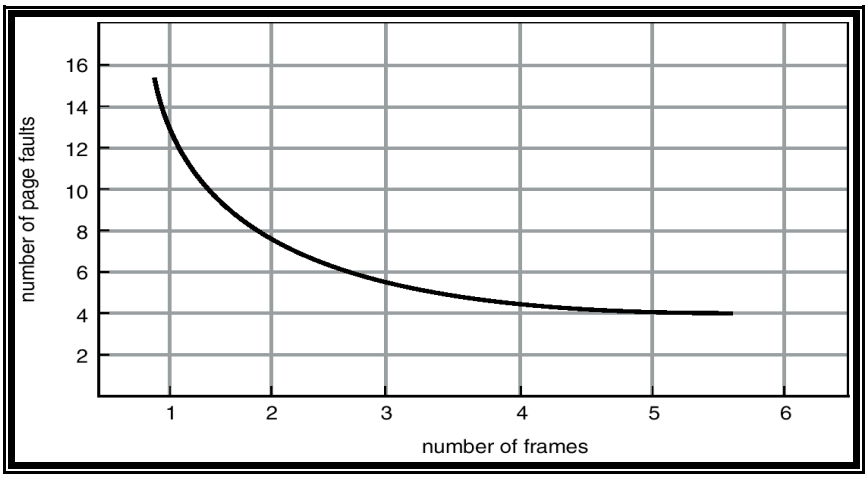

图 6.10 缺页和帧数量关系图

3. FIFO 算法

为了讨论页置换算法，将采用如下引用串：

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

而可用帧的数量为 3。

最简单的页置换算法是 FIFO 算法。FIFO 页置换算法为每个页记录着该页调入内存的时间。当必须置换一页时，将选择最旧的页。注意并不需要记录调入一页的确切时间。可以创建一个 FIFO 队列来管理内存中的所有页。队列中的首页将被置换。当需要调入页时，将它加到队列的尾部。

对于前面的样例引用串，3 个帧开始为空。开始的 3 个引用(7, 0, 1)会引起缺页，将调入这些空帧中。下一个引用(2)置换 7，这是因为页 7 最先调入。由于 0 是下一个引用，但已在

内存中，所以对该引用不会出现缺页。由于页 O 是现在位于内存中的最先被调入的页，对 3 的首次引用导致页 0 被替代。由于这一替代，下一个对 0 的引用会产生缺页，页 1 被页 0 所替代。该进程按图 6.11 所示的方式继续进行，每次有缺页时，都显示哪些页在 3 个帧中。总共有 15 次缺页。

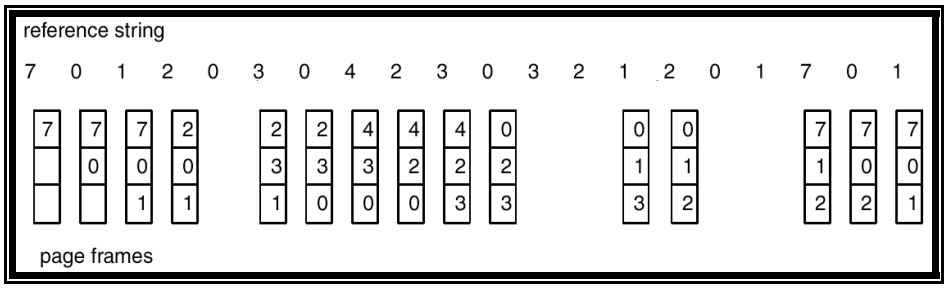

图 6.10 FIFO 页置换算法

FIFO 页置换算法容易理解和实现。但是，其性能并不总是很好。所替代的页可能是很久以前使用的、现已不再使用的初始化模块。另一方面，所替代的页可能包含一个以前初始化的并且不断使用的常用变量。

注意，即使选择替代一个活动页，仍然会正常工作。当换出一个活动页以调入一个新页时，一个缺页几乎马上会要求换回活动页。这样某个页会被替代以将活动页调入内存。因此，一个不好的替代选择增加了缺页率，且减慢了进程执行，但是并不会造成不正确执行。

为了说明与 FIFO 页置换算法相关可能问题，考虑如下引用串：

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

对于可用帧的数量为 3 和 4 的情况，分别统计缺页次数，如图 6.11。

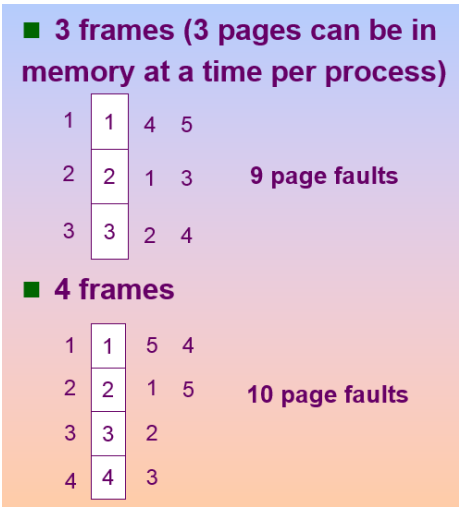

图 6.11 帧的数量为 3 和 4 时的缺页率统计

注意到 4 帧的缺页数比 3 帧的还要大。这种最为令人难以置信的结果称为 Belady 异常 (Belady's anomaly)：对有的页置换算法，缺页率可能会随着所分配的帧数的增加而增加，而原期望为进程增加内存会改善其性能(如图 6.12)。在早期研究中，研究人员注意到这种推测并不总是正确的。因此，发现了 Belady 异常。

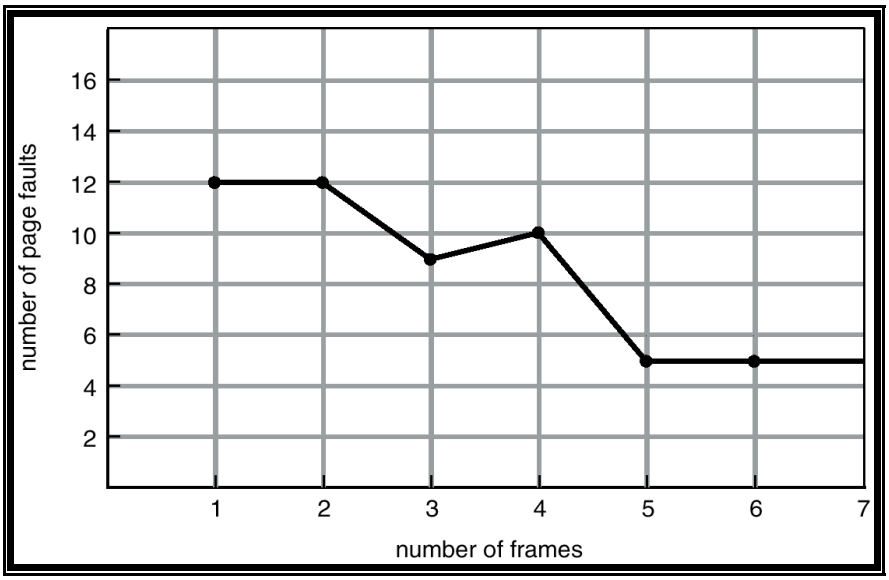

图 6.12 一个引用串的 FIFO 置换的缺页数曲线

4. 最优页置换算法

Belady 异常发现的结果之一是对最优页置换算法(optimal page-replacement algorithm)的搜索。最优页置换算法是所有算法中产生的缺页率最低的，且绝没有 Belady 异常的问题。这种算法确实存在，它被称为 OPT 或 MIN。最优页置换算法选择“未来不再使用的”或“在离当前最远位置上出现的”页被置换。使用这种页置换算法确保对于给定数量的帧会产生最低可能的缺页率。例如，针对前面的引用串样例，最优置换算法会产生 9 个缺页，如图 6.13 所示。头 3 个引用会产生缺页以填满空闲帧。对页 2 的引用会置换页 7，这是因为页 7 直到第 18 次引用时才使用，而页 0 在第 5 次引用时使用，页 1 在第 14 次引用时使用 o。对页 3 的引用，会置换页 1，因为页 l 是位于内存中的 3 个页中最迟引用的页。有 9 个缺页的最优页置换算法要好于有 15 个缺页的 FIFO 算法(如果忽视头 3 个所有算法均会有的缺页，那么最优置换要比 FIFO 置换好一倍)。事实上，没有置换算法能只用 3 个帧且少于 9 个缺页就能处理该引用串。

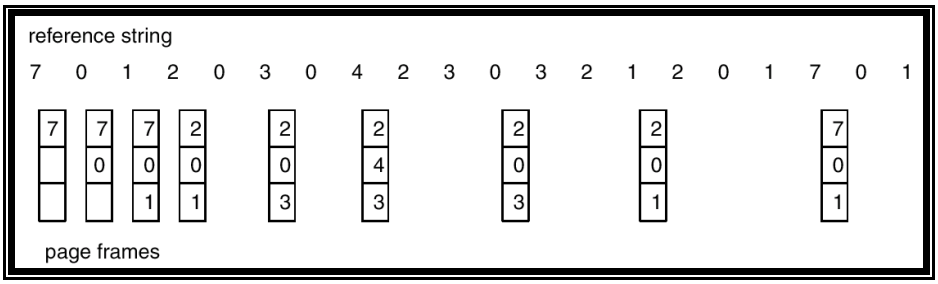

图 6.13 最优置换算法

然而，最优置换算法难以实现，因为需要引用串的未来知识，但这是不可能预测的。。因此，最优算法主要用于比较研究。例如，如果知道一个算法不是最优，但是与最优相比最坏不差于 12.3%，平均不差于 4.7%，那么也是很有用的。

5. LRU 算法

如果最优算法不可行，那么最优算法的近似算法或许成为可能。FIFO 和 OPT 算法(而不是向后看或向前看)的关键区别在于，FIFO 算法使用的是页调入内存的时间，OPT 算法使用的是页将来使用的时间。如果使用离过去最近作为不远将来的近似，那么可置换最长时间没有

使用的页(见图 6.14)，这种方法称为最近最少使用算法(least-recently-used algorithm，LRU)。

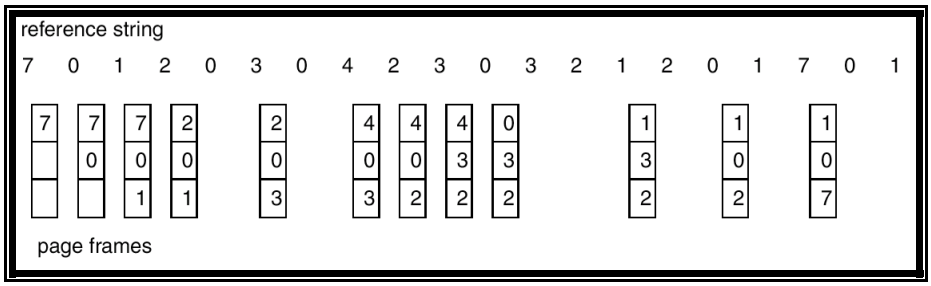

图 6.14 LRU 页置换算法

对引用串样例，采用 LRU 置换的结果如图 6.14 所示。LRU 算法产生 12 次缺页。注意，头 5 个错误与最优算法一样。然而，当出现对页 4 的引用时，LRU 算法知道页 2 最近最少使用。因此，LRU 算法置换页 2;并不知道页 2 会马上要用。接着，当页 2 出错时，LRU 算法会置换页 3，这是因为位于内存的 3 个页中，页 3 最近最少使用。虽然存在这些问题，有 12 个缺页的 LRU 置换仍然要比有 15 个缺页的 FIFO 置换要好。

最优置换和 LRU 置换都没有 Be1ady 异常。这两个都属于同一类算法，称为栈算法(stack algorithm)，都绝不可能有 Be1ady 异常。核算法可以证明为:对于帧数为 n 的内存页集合是对于帧数为 n +1 的内存页集合的子集。对于 LRU 算法，如果内存页的集合为最近引用的页，那么对于帧的增加，这 n 页仍然为最近引用的页，所以也仍然在内存中。

LRU 选择内存中最久未使用的页面被置换。这是局部性原理的合理近似，性能接近最佳算法。但由于需要记录页面使用时间的先后关系，硬件开销较大。它的问题是为页帧确定一个排序序列，这个序列按页帧上次使用的时间来定。有两种可行实现:

计数器：最为简单的情况是，为每个页表项关联一个使用时间域，并为 CPU 增加一逻辑时钟或计数器。对每次内存引用，计数器都会增加。每次内存引用时，时钟寄存器的内容会被复制到相应页所对应页表项的使用时间域内。用这种方式就得到每页的最近引用时间。置换具有最小时间的页。这种方案需要搜索页表以查找 LRU 页，且每次内存访问都要写入内存(到页表的使用时间域)。在页表改变时(因 CPU 调度)也必须保持时间。必须考虑时钟溢出。

栈：实现 LRU 置换的另一个方法是采用页码枝。每当引用一个页，该页就从校中删除并放在顶部。这样，栈顶部总是最近使用的页，枝底部总是 LRU 页(图 6.15)。由于必须从栈中部删除工页，所以该校可实现为具有头指针和尾指针的双向链表。这样，删除一页并放在栈顶部在最坏情况下需要改变 6 个指针。虽说每个更新有点费时，但是置换不需要搜索：尾指针指向枝底部，就是 LRU 页。对于用软件或微代码的 LRU 置换的实现，这种方法十分合适。

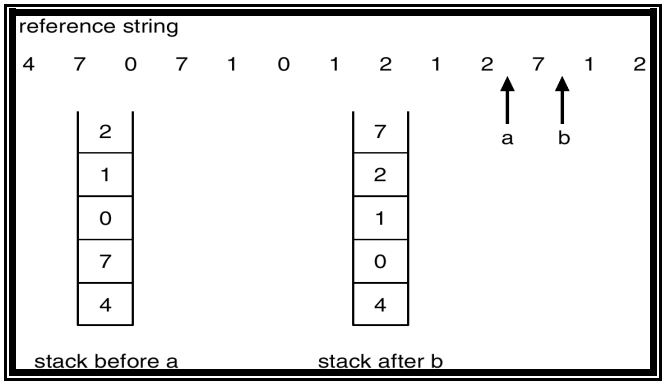

图 6.15 用找来记录最近使用的页

如果只有标准TLB寄存器而没有其他硬件支持，那么这两种LRU实现都是不可能的。每次内存引用都必须更新时钟域或拢。如果对每次引用都采用中断，以允许软件更新这些数据结构，那么它会使内存引用慢至少10倍，进而使用户进程运行慢10倍。几乎没有系统可以容忍如此程度的内存管理的开销。

6. 近似LRU算法

很少有计算机系统能提供足够的硬件来支持真正的LRU页置换。有的系统不提供任何支持，因此必须使用其他置换算法(如FIFO算法)。然而，许多系统都通过引用位方式提供一定的支持。页表内的每项都关联着一个引用位(或叫访问位，reference bit)。每当引用一个页时(无论是对页的字节进行读或写)，相应页表的引用位就被硬件置位。

开始，操作系统会将所有引用位都清零。随着用户进程的执行，与引用页相关联的引用位被硬件置位(置为1)。之后，通过检查引用位，能够确定哪些页使用过而哪些页未使用过。虽然不知道使用顺序，但是知道哪些页用过而哪些页未用过。这信息是许多近似LRU页置换算法的基础。

(1) 附加引用位算法

通过在规定时间间隔里记录引用位，可以获得额外顺序信息。可以为位于内存内的每个表中的页保留一个8位的字节。在规定时间间隔(如，每100ms)内，时钟定时器产生中断并将控制转交给操作系统。操作系统把每个页的引用位转移到其8位字节的高位，而将其他位向右移一位，并抛弃最低位。这些8位移位寄存器包含着该页在最近8个时间周期内的使用情况。例如，如果移位寄存器含有00000000，那么该页在8个时间周期内没有使用：如果移位寄存器的值为11111111，那么该页在过去每个周期内都至少使用过一次。具有值为11000100的移位寄存器的页要比值为01110111的页更为最近使用。如果将这8位字节作为无符号整数，那么具有最小值的页为LRU页，且可以被置换。注意这些数字并不唯一。可以置换所有具存最小值的页，或在这些页之间采用FIFO来选择置换。

当然，历史位的数量口可以修改，可以选择(依赖于可用硬件)以尽可能快地更新。在极端情况下，数量可降为0，即只有引用位本身。这种算法称为二次机会页置换算法(second-chance page-replacement algorithm)。

(2)二次机会算法(也称为时钟算法，clock算法)

二次机会置换的基本算法是FIFO置换算法。当要选择一个页时，检查其引用位。如果其值为0，那么就直接置换该页。如果引用位为1，那么就给该页第二次机会，并选择下一个FIFO页。当一个页获得第二次机会时，其引用位清零，且其到达时间设为当前时间。因此，获得第二次机会的页在所有其他页置换(或获得第二次机会)之前，是不会被置换的。另外，如果一个页经常使用以致其引用位总是被设置，那么它就不会被置换。

一种实现二次机会算法的方法是采用循环队列。用一个指针表示下次要置换哪一页。当需要一个帧时，指针向前移动直到找到一个引用位为0的页。在向前移动时，它将清除引用位(见图6.16)。一旦找到牺牲页，就置换该页，新页就插入到循环队列的该位置。注意：在最坏情况下，所有位均已设置，指针会遍历整个循环队列，以便给每个页第二次机会。它将清除所再引用位后再选择页来置换。这样，如果所有位均已设置，那么二次机会置换就变成了FIFO置换。

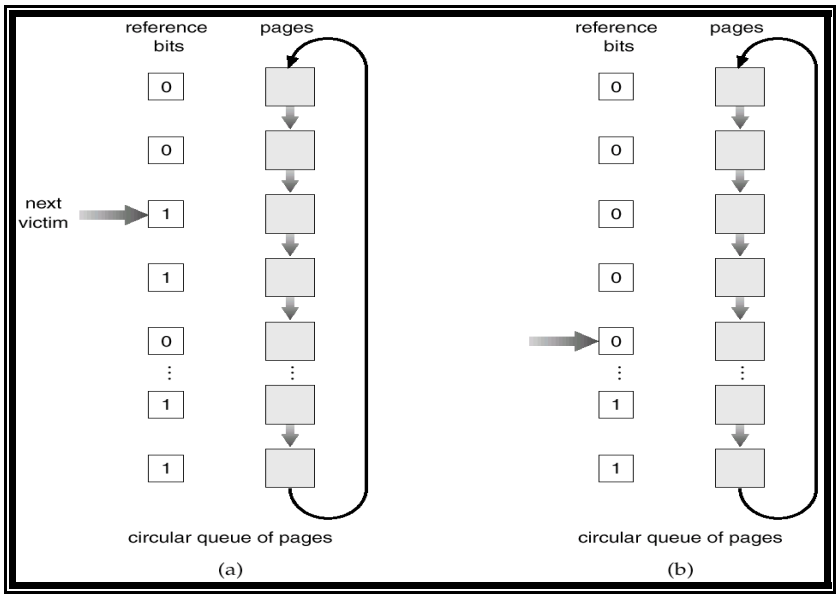

图 6.16 二次机会(时钟)页置换算法

(3) 增强型二次机会算法

通过将引用位和修改位作为一有序对来考虑，可以改进二次机会算法。采用这两个位，再下面四种可能类型:

(0, 0) 最近没有使用且也没有修改——用于置换的最佳页。

(0, 1) 最近没有使用但修改过——不是很好，因为在置换之前需要将页写出到磁盘。

(1,0) 最近使用过但没有修改_它有可能很快又要被使用。

(1,1) 最近使用过且修改过_它有可能很快又要被使用，且置换之前需要将页写出到磁盘。

每个页都属于这四种类型之一。当页需要置换时，可使用时钟算法，不是检查所指页的引用位是否设置，而是检查所指页属于哪个类型。四种类型的页淘汰次序 (0, 0) ⇒ (0, 1) ⇒ (1, 0) ⇒ (1, 1)。注意在找到要置换页之前，可能要多次搜索整个循环队列。

这种方法与简单时钟算法的主要差别是这里给那些已经修改过的页以更高的级别，从而降低了所需 I/O 的数量。此算法在 Macintosh 系统中使用。

7. 基于计数的页置换

还有许多其他算法可用于页置换。例如，可以为每个页保留一个用于记录其引用次数的计数器，并可形成如下两个方案。

(1) 最不经常使用页置换算法(least frequently used page-replacement algorithm, LFU)要求置换计数最小的页。这种选择的理由是活动页应该有更大的引用次数。这种算法会产生如下问题:一个页在进程开始时使用很多，但以后就不再使用。由于其使用过很多，所以它有较大次数，所以即使不再使用仍然会在内存中。解决方法之一是定期地将次数寄存器右移一位，以形成指数衰减的平均使用次数。

(2) 最常使用页置换算法(most frequently used page-replacement algorithm, MFU)是基于如下理论：具有最小次数的页可能刚刚调进来，且还没有使用。

MFU 和 LFU 置换都不常用。这两种算法的实现都很费时，且并不能很好地近似 OPT 置换算法。