# 概述

页高速缓存（cache）是Linux内核实现磁盘缓存。它主要用来减少对磁盘的I/O操作。具体地讲，是通过把磁盘中的数据缓存到物理内存中，把对磁盘的访问变为对物理内存的访问。

## 作用

磁盘高速缓存之所以在任何现代操作系统中尤为重要有两个主要因素：第一，访问磁盘的速度要远远低于（差好几个数量级）访问内存的速度——ms和ns的差距，因此，从内存访问数据比从磁盘访问速度更快，若从处理器的L1和L2高速缓存访问则更快。第二，数据一旦被访问，就很有可能在短时间内再次被访问到。

## 局部性原理

这种在短时间内集中访问同一片数据的原理称作临时局部原理（temporal locality）。临时局部原理能保证：如果在第一次访问数据时缓存它，那就极有可能在短期内再次被高速缓存命中（访问到高速缓存中的数据）。正是由于内存访问要比磁盘访问快得多，再加上数据一次被访问后更可能再次被访问的特点，所以磁盘的内存缓存将给系统存储性能带来质的飞跃。

# 缓存手段

页高速缓存是由**内存**中的**物理页面**组成的，**其内容对应磁盘上的物理块**。页高速缓存大小能够动态调整——它可以通过占用空闲内存以扩张大小，也可以自我收缩以缓解内存使用压力。

当内核开始一个读操作（比如进程发起一个read()系统调用），它首先会检查需要的数据是否在页高速缓存中。如果在，则放弃访问磁盘，而直接从内存中读取。这个行为称作缓存命中。如果数据没有在缓存中，称为缓存未命中，那么内核必须调度块I/O操作从磁盘去读取数据。然后内核将读来的数据放入页缓存中，于是任何后续相同的数据读取都可以命中缓存了。

注意，系统并不一定要将整个文件都缓存。缓存可以持有某个文件的全部内容，也可以存储另一些文件的一页或者几页。到底该缓存谁取决于谁被访问到。

## 写缓存

通常来讲，缓存一般被实现成三种策略：

第一种称为不缓存（nowrite），也就是说高速缓存不去缓存任何写操作。当对一个缓存中的数据片进行写时，将直接跳过缓存，写到磁盘，同时也使缓存中的数据失效。那么如果后续读操作进行时，需要再重新从磁盘中读取数据。

不过这种策略很少使用，因为该策略不但不去缓存写操作，而且需要额外费力去使缓存数据失效。

第二种策略，写操作将自动更新内存缓存，同时也更新磁盘文件。这种方式，通常称为写透缓存（write-through cache），因为写操作会立刻穿透缓存到磁盘中。这种策略对保持缓存一致性很有好处——缓存数据时刻和后备存储保持同步，所以不需要让缓存失效，同时它的实现也最简单。

第三种策略，也就是Linux所采用的，称为**“回写”**。在这种策略下，程序执行写操作直接写到缓存中，后端存储不会立刻直接更新，而是将页高速缓存中被写入的页面标记为“脏”，并且被加入到脏页链表中。然后由一个进程（回写进程）周期将脏页链表中的页写回到磁盘，从而让磁盘中的数据和内存中最终一致。最后清理“脏页”标识。

回写策略通常认为要好于写透策略。因为通过延迟写磁盘，方便在以后的时间内合并更多的数据和再一次刷新。当然，其代价是实现复杂度高。

注：参考Redis缓存更新策略和MySQL的日志更新。

## 缓存回收

Linux的缓存回收是通过选择干净页（不脏）进行简单替换。如果缓存中没有足够的干净页面，内核将强制地进行回写操作，以腾出更多的干净可用页。最难的事情在于决定什么页应该回收。理想的回收策略应该是回收那些以后最不可能使用的页面。理想的回收策略称为预测算法。

缓存回收策略：

### LRU

缓存回收策略通过所访问的数据特性，尽量追求预测效率。最成功的算法（特别是对于通用目的的页高速缓存）称作最近最少使用算法，简称LRU。

LRU回收策略需要追踪每个页面的访问踪迹（或者至少按照访问时间为序的页链表），以便回收最老时间戳的页面（或者回收排序链表头所指的页面）。

该策略的良好效果源自于缓存的数据越久未被访问，则越不大可能近期再次被访问，而最近被访问的最有可能再次被访问。但是，对于许多文件被访问一次，再不被访问的情景，LRU尤其失败。将这些页面放在LRU链表的顶端显然不是最优，当然，内核无法知道一个问卷只会被访问一次，但是它却知道过去访问了多少次。

注：在MySQL中LRU算法与这个有点区别，为了防止大面积的缓存失效，做了调整（5/8LRU）。

### 双链策略

Linux实现的是一个修改过的LRU，也称为双链策略。和以前不同，Linux维护的不再是一个LRU链表，而是维护两个链表：活跃链表和非活跃链表。处于活跃链表上的页面被认为是“热”的且不会被换出，而在非活跃链表上的页面则是可以被换出的。在活跃链表中的页面必须在其被访问时就处于非活跃链表中。两个链表都被伪LRU规则维护：页面从尾部加入，从头部移除，如同队列。两个链表需要保持平衡——如果活跃链表变得过多而超过了非活跃链表，那么活跃链表的头页面将被重新回到非活跃链表中，以便能再被回收。双链表策略解决了传统LRU算法中对仅一次访问的窘境。而且也更加简单的实现了伪LRU语义。这种双链表方式也称作LRU/2.更普遍的是n个链表，故称LUR/n。

注：传统的LRU将页刷新出去之后就不在链表中了，但是双链策略可以将页从活跃链表转移到非活跃链表中。

### LFU

# Linux页高速缓存

## address\_space对象

## address\_space操作

## 基树

因为在任何页I/O操作前内核都要检查页是否已经在页高速缓存中了，所以这种频繁进行的检查必须迅速、高效，否则搜索和检查页高速缓存的开销可能抵消页高速缓存带来的好处（至少在缓存命中率较低的时候，搜索的开销足以抵消以内存代替磁盘进行检索数据带来的好处）。

注：在KVM虚拟化中，定位资源池中的虚机是否已经存在可以使用radix基树。

## 以前的页散列表

在2.6版本以前，内核页高速缓存不是通过基树检索，而是通过一个维护了系统中所有页的全局散列表进行检索。对于给定的一个键值，该散列表会返回一个双链表的入口对应于这个所给定的值。如果需要的页贮存在缓存中，那么链表中的一项就会与其对应。否则，页就不在页面高速缓存中，散列函数返回NULL。

全局散列表主要存在四个问题：

1. 由于使用单个的全局锁保护散列表，所以即使在中等规模的机器中，锁的争用情况也会相当严重，造成性能受损；
2. 由于散列表需要包含所有页高速缓存中的页，可是搜索需要的只是和当前文件相关的那些页，所以散列表包含的页面相比搜索需要的页面要大很多；
3. 如果散列搜索失败（也就是给定的页不在页高速缓存中），执行速度要比希望的慢很多，这是因为检索必须遍历指定散列键值对应的整个链表；
4. 散列表比其他方法会消耗更多的内存。

2.6版本内核中引入基于基树的页高速缓存解决这些问题。

# 缓冲区高速缓存

独立的磁盘块通过块I/O缓冲也要被存入页高速缓存。

块I/O操作一次操作一个单独的磁盘块。普通的块I/O操作是读写i节点。内核提供了bread()函数实现从磁盘读一个块的底层操作。通过缓存，磁盘块映射到它们相关的内存页，并缓存到页高速缓存中。

在2.4内核版本前，缓冲和页面高速缓存不是统一的。在内核中，存在两个独立的磁盘缓存：页高速缓存和缓冲区高速缓存。前者缓存页面，后者缓存缓冲区，这两个缓存并没有统一。一个磁盘块可以同时存于两个缓存中，这导致必须同步操作两个缓冲中的数据，而且浪费了内存，去存储重复的缓存项。现在，只有一个磁盘缓存，即页高速缓存。虽然如此，内核仍然需要在内存中使用缓冲来表示磁盘块，幸好，缓冲是页映射块的，所以它正好在页高速缓存中。

# flusher线程