**主要内容：**

* 缓存简介
* 页高速缓存
* 页回写

### ****1. 缓存简介****

#### **1.1 原理**

之所以通过缓存能提高I/O性能是基于以下2个重要的原理：

1. CPU访问内存的速度远远大于访问磁盘的速度（访问速度差距不是一般的大，差好几个数量级）
2. 数据一旦被访问，就有可能在短期内再次被访问（临时局部原理）

#### **1.2 策略**

缓存的创建和读取没什么好说的，无非就是检查缓存是否存在要创建或者要读取的内容。

但是写缓存和缓存回收就需要好好考虑了，这里面涉及到「缓存内容」和「磁盘内容」同步的问题。

**1.2.1 「写缓存」常见的有3种策略**

1. 不缓存(nowrite) :: 也就是不缓存写操作，当对缓存中的数据进行写操作时，直接写入磁盘，同时使此数据的缓存失效
2. 写透缓存(write-through) :: 写数据时同时更新磁盘和缓存
3. 回写(copy-write or write-behind) :: 写数据时直接写到缓存，由另外的进程(回写进程)在合适的时候将数据同步到磁盘

**1.2.2 「缓存回收」的策略**

1. 最近最少使用(LRU) :: 每个缓存数据都有个时间戳，保存最近被访问的时间。回收缓存时首先回收时间戳较旧的数据。
2. 双链策略(LRU/2) :: 基于LRU的改善策略。具体参见下面的补充说明

**补充说明(双链策略):**

双链策略其实就是 LRU(Least Recently Used) 算法的改进版。

它通过2个链表(活跃链表和非活跃链表)来模拟LRU的过程，目的是为了提高页面回收的性能。

页面回收动作发生时，从非活跃链表的尾部开始回收页面。

双链策略的关键就是页面如何在2个链表之间移动的。

双链策略中，每个页面都有2个标志位，分别为

PG\_active - 标志页面是否活跃，也就是表示此页面是否要移动到活跃链表

PG\_referenced - 表示页面是否被进程访问到

页面移动的流程如下：

1. 当页面第一次被被访问时，PG\_active 置为1，加入到活动链表
2. 当页面再次被访问时，PG\_referenced 置为1，此时如果页面在非活动链表，则将其移动到活动链表，并将PG\_active置为1，PG\_referenced 置为0
3. 系统中 daemon 会定时扫描活动链表，定时将页面的 PG\_referenced 位置为0
4. 系统中 daemon 定时检查页面的 PG\_referenced，如果 PG\_referenced=0，那么将此页面的 PG\_active 置为0，同时将页面移动到非活动链表

参考：[Linux 2.6 中的页面回收与反向映射](http://www.ibm.com/developerworks/cn/linux/l-cn-pagerecycle/index.html" \t "https://www.cnblogs.com/wang_yb/p/_blank)

### ****2. 页高速缓存****

故名思义，页高速缓存中缓存的最小单元就是内存页。

#### **2.1 简介**

页高速缓存缓存的是具体的物理页面，与前面章节中提到的虚拟内存空间(vm\_area\_struct)不同，假设有进程创建了多个 vm\_area\_struct 都指向同一个文件，

那么这个 vm\_area\_struct 对应的 页高速缓存只有一份。

也就是磁盘上的文件缓存到内存后，它的虚拟内存地址可以有多个，但是物理内存地址却只能有一个。

下面通过分析内核中的相应的结构体，来了解内核是如何提高 I/O性能的。

#### **2.2 实现**

实现页高速缓存的最重要的结构体要算是 address\_space ，在 <linux/fs.h> 中

struct address\_space {

struct inode \*host; /\* 拥有此 address\_space 的inode对象 \*/

struct radix\_tree\_root page\_tree; /\* 包含全部页面的 radix 树 \*/

spinlock\_t tree\_lock; /\* 保护 radix 树的自旋锁 \*/

unsigned int i\_mmap\_writable;/\* VM\_SHARED 计数 \*/

struct prio\_tree\_root i\_mmap; /\* 私有映射链表的树 \*/

struct list\_head i\_mmap\_nonlinear;/\* VM\_NONLINEAR 链表 \*/

spinlock\_t i\_mmap\_lock; /\* 保护 i\_map 的自旋锁 \*/

unsigned int truncate\_count; /\* 截断计数 \*/

unsigned long nrpages; /\* 总页数 \*/

pgoff\_t writeback\_index;/\* 回写的起始偏移 \*/

const struct address\_space\_operations \*a\_ops; /\* address\_space 的操作表 \*/

unsigned long flags; /\* gfp\_mask 掩码与错误标识 \*/

struct backing\_dev\_info \*backing\_dev\_info; /\* 预读信息 \*/

spinlock\_t private\_lock; /\* 私有 address\_space 自旋锁 \*/

struct list\_head private\_list; /\* 私有 address\_space 链表 \*/

struct address\_space \*assoc\_mapping; /\* 缓冲 \*/

struct mutex unmap\_mutex; /\* 保护未映射页的 mutux 锁 \*/

} \_\_attribute\_\_((aligned(sizeof(long))));

**补充说明：**

1. inode - 如果 address\_space 是由不带inode的文件系统中的文件映射的话，此字段为 null
2. page\_tree - 这个树结构很重要，它保证了页高速缓存中数据能被快速检索到，脏页面能够快速定位。
3. i\_mmap - 根据 vm\_area\_struct，能够快速的找到关联的缓存文件(即 address\_space)，前面提到过， address\_space 和 vm\_area\_struct 是 一对多的关系。
4. 其他字段主要是提供各种锁和辅助功能

此外，对于这里出现的一种新的数据结构 radix 树，进行简要的说明。

radix树通过long型的位操作来查询各个节点， 存储效率高，并且可以快速查询。

linux中 radix树相关的内容参见： include/linux/radix-tree.h 和 lib/radix-tree.c

### ****3. 页回写****

由于目前linux内核中对于「写缓存」采用的是第3种策略，所以回写的时机就显得非常重要，回写太频繁影响性能，回写太少容易造成数据丢失。

#### **3.1 简介**

linux 页高速缓存中的回写是由内核中的一个线程(flusher 线程)来完成的，flusher 线程在以下3种情况发生时，触发回写操作。

1. 当空闲内存低于一个阀值时

    空闲内存不足时，需要释放一部分缓存，由于只有不脏的页面才能被释放，所以要把脏页面都回写到磁盘，使其变成干净的页面。

2. 当脏页在内存中驻留时间超过一个阀值时

   确保脏页面不会无限期的驻留在内存中，从而减少了数据丢失的风险。

3. 当用户进程调用 sync() 和 fsync() 系统调用时

   给用户提供一种强制回写的方法，应对回写要求严格的场景。

页回写中涉及的一些阀值可以在 /proc/sys/vm 中找到

下表中列出的是与 pdflush(flusher 线程的一种实现) 相关的一些阀值

|  |  |
| --- | --- |
| **阀值** | **描述** |
| dirty\_background\_ratio | 占全部内存的百分比，当内存中的空闲页达到这个比例时，pdflush线程开始回写脏页 |
| dirty\_expire\_interval | 该数值以百分之一秒为单位，它描述超时多久的数据将被周期性执行的pdflush线程写出 |
| dirty\_ratio | 占全部内存的百分比，当一个进程产生的脏页达到这个比例时，就开始被写出 |
| dirty\_writeback\_interval | 该数值以百分之一秒未单位，它描述pdflush线程的运行频率 |
| laptop\_mode | 一个布尔值，用于控制膝上型计算机模式 |

#### **3.2 实现**

flusher线程的实现方法随着内核的发展也在不断的变化着。下面介绍几种在内核发展中出现的比较典型的实现方法。

1. 膝上型计算机模式

这种模式的意图是将硬盘转动的机械行为最小化，允许硬盘尽可能长时间的停滞，以此延长电池供电时间。

该模式通过 /proc/sys/vm/laptop\_mode 文件来设置。(0 - 关闭该模式  1 - 开启该模式)

2. bdflush 和 kupdated (2.6版本前 flusher 线程的实现方法)

bdflush 内核线程在后台运行，系统中只有一个 bdflush 线程，当内存消耗到特定阀值以下时，bdflush 线程被唤醒

kupdated 周期性的运行，写回脏页。

**bdflush 存在的问题：**

整个系统仅仅只有一个 bdflush 线程，当系统回写任务较重时，bdflush 线程可能会阻塞在某个磁盘的I/O上，

导致其他磁盘的I/O回写操作不能及时执行。

3. pdflush (2.6版本引入)

pdflush 线程数目是动态的，取决于系统的I/O负载。它是面向系统中所有磁盘的全局任务的。

**pdflush 存在的问题：**

pdflush的数目是动态的，一定程度上缓解了 bdflush 的问题。但是由于 pdflush 是面向所有磁盘的，

所以有可能出现多个 pdflush 线程全部阻塞在某个拥塞的磁盘上，同样导致其他磁盘的I/O回写不能及时执行。

4. flusher线程 (2.6.32版本后引入)

flusher线程改善了上面出现的问题：

首先，flusher 线程的数目不是唯一的，这就避免了 bdflush 线程的问题

其次，flusher 线程不是面向所有磁盘的，而是每个 flusher 线程对应一个磁盘，这就避免了 pdflush 线程的问题