

Linux内核文件Cache 机制

<http://www.ilinuxkernel.com>

本文链接：<http://ilinuxkernel.com/?p=1700>

目 录Table of Contents

1	概述	5
2	Cache重要数据结构和函数	7
2.1	address_space	7
2.2	address_space_operations	10
2.3	find_get_page ()	11
2.4	add_to_page_cache ()	12
2.5	remove_from_page_cache ()	13
3	Linux内核预读机制	14
3.1	file_ra_state数据结构	15
3.2	do_generic_file_read ()	16
3.3	page_cache_sync_readahead ()	17
3.4	page_cache_async_readahead ()	18
3.5	ondemand_readahead ()	19
3.6	ra_submit ()	22
3.7	__do_page_cache_readahead ()	22
3.8	read_pages ()	24
3.9	mpage_readpages ()	25
4	文件读写I/O流程与Cache机制	26
4.1	读文件过程	26
4.1.1	数据不在页面Cache中	27
4.1.2	数据在页面Cache中	28
4.2	写文件过程	29

图目录 List of Figures

图1 内核中块设备操作流程	6
---------------------	---

表目录 List of Tables

表1 address_space对象成员变量含义	8
表2 file_ra_state数据结构成员变量含义.....	16

1 概述

在我们使用Linux时，使用**free**命令观察系统内存使用情况，如下面空间内存为66053100k。可能很多人都遇到过一个问题，发现随着时间的推移，内存的**free**越来越小，而**cached**越来越大；于是就以为是不是自己的程序存在内存泄漏，或者是硬件、操作系统出了问题？显然，从这里看不出用户程序是否有内存泄漏，也不是内核有Bug或硬件有问题。原因是内核的文件Cache机制。实际上文件Cache的实现是页面Cache，本文后续都以页面Cache来描述。

```
[root@localhost ~]# free
              total        used        free      shared    buffers     cached
Mem:      66053100    1727572    64325528           0     242492     409440
-/+ buffers/cache:    1075640    64977460
Swap:      2097144           0     2097144
```

当应用程序需要读取文件中的数据时，操作系统先分配一些内存，将数据从存储设备读入到这些内存中，然后再将数据分发给应用程序；当需要往文件中写数据时，操作系统先分配内存接收用户数据，然后再将数据从内存写到磁盘上。**文件Cache管理**指的就是对这些由操作系统内核分配，并用来存储文件数据的内存管理。

在大部分情况下，内核在读写磁盘时都先通过页面Cache。若页面不在Cache中，新页加入到页面Cache中，并用从磁盘上读来的数据来填充页面。如果内存有足够的内存空间，该页可以在页面Cache长时间驻留，其他进程再访问该部分数据时，不需要访问磁盘。这就是**free**命令显示内核**free**值越来越小，**cached**值越来越大的原因。

同样，在把一页数据写到块设备之前，内核首先检查对应的页是否已经在页面Cache中；如果不在，就在页面Cache增加一个新页面，并用要写到磁盘的数据来填充。数据的I/O传输并不会立即开始执行，而是会延迟几秒左右；这样进程就有机会进一步修改写到磁盘的数据。

内核的代码和数据结构不必从磁盘读，也不必写入磁盘。因此页面Cache可能是下面的类型：

- 含有普通文件数据的页；

- 含有目录的页；
- 含有直接从块设备文件（跳过文件系统层）读出的数据页；
- 含有用户态进程数据的页，但页中的数据已被交换到磁盘；
- 属于特殊文件系统的页，如进程间通信中的特殊文件系统shm。

图1是块设备I/O操作流程，从图中我们可以看出具体文件系统（如ext3/ext4、xfs等），负责在文件Cache和存储设备之间交换数据，位于具体文件系统之上的虚拟文件系统VFS负责在应用程序和文件 Cache 之间通过read（）/write（）等接口交换数据。

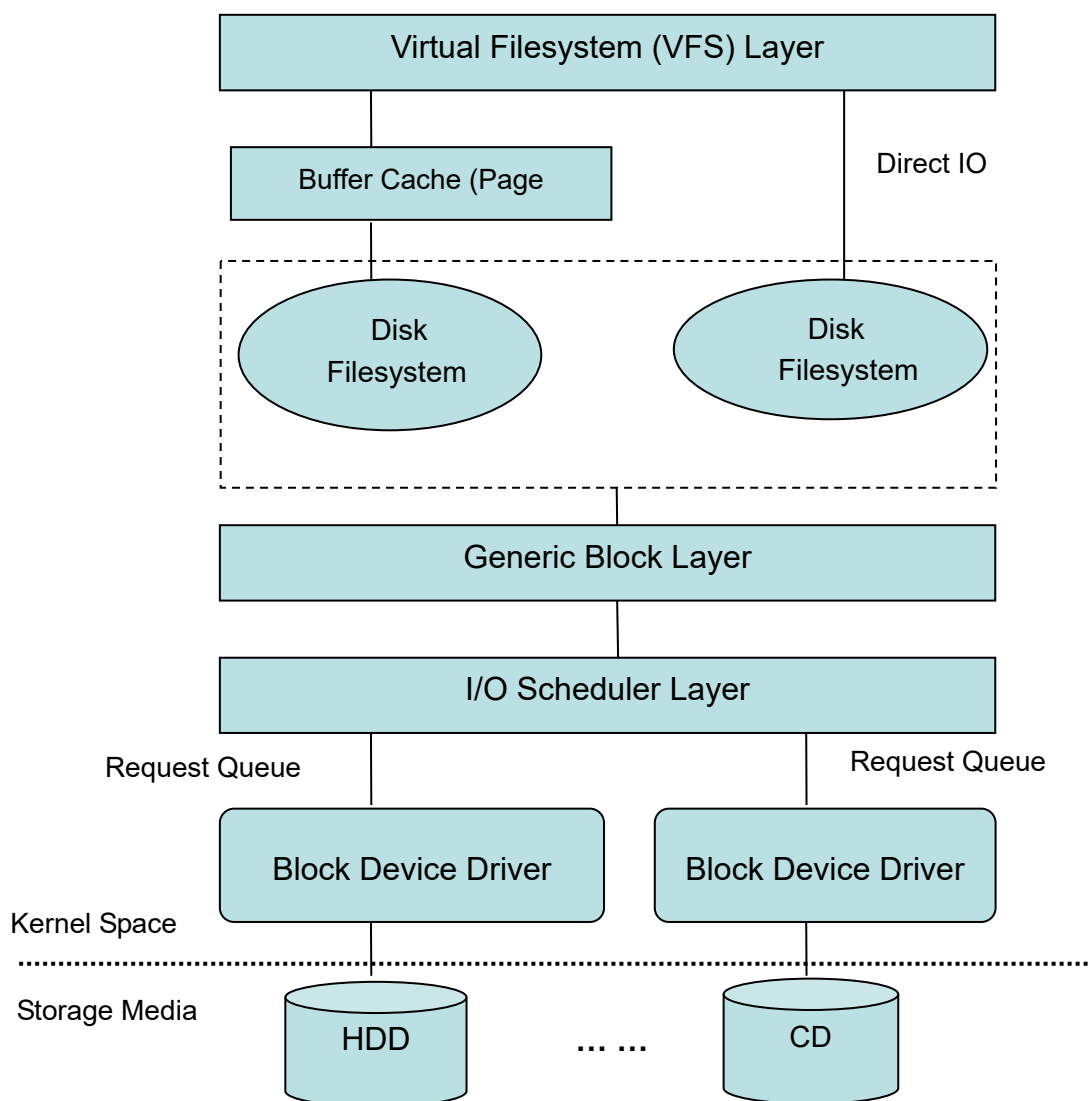


图1 内核中块设备操作流程

页面Cache中的每页所包含的数据是属于某个文件，这个文件（准确地说是文件的inode）就是该页的拥有者。事实上，所有的read（）和write（）都依赖于页面Cache；唯

一的例外是当进程打开文件时，使用了O_DIRECT标志，在这种情况下，页面Cache被跳过，且使用了进程用户态地址空间的缓冲区。有些数据库应用程序使用O_DIRECT标志，这样他们可以使用自己的磁盘缓冲算法。

内核页面Cache的实现主要为了满足下面两种需要：

- 快速定位含有给定所有者相关数据的特定页。为了尽可能发挥页面Cache的优势，查找过程必须是快速的。
- 记录在读或写页中的数据时，应该如何处理页面Cache中的每个页。例如，从普通文件、块设备文件或交换区读一个数据页，必须用不同的方式；这样内核必须根据页面拥有者来选择正确的操作。

显然，页面Cache中的数据单位是整页数据。当然一个页面中的数据在磁盘上不必是相邻的，这样页面就不能用设备号和块号来识别。取而代之的是，Cache中的页面识别是通过拥有者和拥有者数据中的索引，通常是inode和相应文件内的偏移量。

文件Cache是文件数据在内存中的副本，因此文件Cache管理与内存管理系统和文件系统都相关：一方面文件 Cache 作为物理内存的一部分，需要参与物理内存的分配回收过程，另一方面文件Cache中的数据来源于存储设备上的文件，需要通过文件系统与存储设备进行读写交互。从操作系统的角度考虑，文件Cache可以看做是内存管理系统与文件系统之间的联系纽带。因此，文件 Cache管理是操作系统的一个重要组成部分，它的性能直接影响着文件系统和内存管理系统的性能。

2 Cache重要数据结构和函数

2.1 address_space

页面Cache的核心数据结构是address_space，定义在include/linux/fs.h中。

address_space结构体嵌入在拥有该页面的inode对象中。在Cache中，可能有多个页面同属于一个inode，这样他们就可能指向同一个address_space对象。同时，通过该对象将拥有者的页面和在这些页面上的操作方法联系起来。address_space对象中成员变量含义如表1所示。

00664: **struct address_space {**

```

00665:  struct inode      *host;      /* owner: inode, block_device */
00666:  struct radix_tree_root page_tree; /* radix tree of all pages */
00667:  spinlock_t      tree_lock;    /* and lock protecting it */
00668:  unsigned int     i_mmap_writable; /* count VM_SHARED mappings */
00669:  struct prio_tree_root i_mmap; /* tree of private and shared mappings */
00670:  struct list_head i_mmap_nonlinear; /* list VM_NONLINEAR mappings */
00671:  spinlock_t      i_mmap_lock; /* protect tree, count, list */
00672:  unsigned int     truncate_count; /* Cover race condition with truncate */
00673:  /* Protected by tree_lock together with the radix tree */
00674:  unsigned long    nrpages;     /* number of total pages */
00675:  pgoff_t          writeback_index; /* writeback starts here */
00676:  const struct address_space_operations *a_ops; /* methods */
00677:  unsigned long    flags;       /* error bits/gfp mask */
00678:  struct backing_dev_info *backing_dev_info; /* device readahead, etc */
00679:  spinlock_t      private_lock; /* for use by the address_space */
00680:  struct list_head private_list; /* ditto */
00681:  struct address_space *assoc_mapping; /* ditto */
00682: } __attribute__((aligned(sizeof(long))));

```

每个页面描述符（struct page）包含两个成员变量mapping和index，它们是用来链接页面到页面Cache。mapping指向inode（拥有该页面）的地址_space对象；index是拥有者“地址空间”（这种情况下可以地址空间理解为磁盘上的文件）内的偏移量，单位大小是页。当在页面Cache中查找某个页时，就是使用成员变量mapping和index。

表1 address_space对象成员变量含义

类型	字段	含义
struct inode *	host	指向拥有该对象的节点
struct radix_tree_root	page_tree	表示拥有者的页基树(radix tree)的根
spinlock_t	tree_lock	保护基树的自旋锁
unsigned int	i_mmap_writable	地址空间中共享内存映射的个数
struct prio_tree_root	i_mmap	radix优先搜索树的根
struct list_head	i_mmap_nonlinear	地址空间中非线性内存区的链表
spinlock_t	i_mmap_lock	保护radix优先搜索树的自旋锁
unsigned int	truncate_count	截断文件时使用的顺序计数器
unsigned long	nrpages	所有者的页总数
unsigned long	writeback_index	最后一次回写操作所作用的页索引
struct address_space_operations *	a_ops	对所有者页进行操作的方法
unsigned long	flags	错误位和内存分配标志
struct backing_dev_info *	backing_dev_info	指向拥有者的数据所在块设备的backing_dev_info
spinlock_t	private_lock	通常是管理private_list链表时使用的自旋锁
struct list head	private_list	通常是与索引节点相关的间接快的脏缓冲区链表
struct address_space *	assoc_mapping	通常指向间接块所在块设备的address_space对象

注意，在某些情况下，页面Cache可能包含磁盘上数据多份拷贝。因为一个普通文件的

4K数据，可以用以下方式访问：

- 读取文件；这样页面中的数据拥有者是这个普通文件的inode。
- 直接读取块设备文件（如/dev/sda1）；这样页面中的数据拥有者是块设备文件的主inode。

这样同一份磁盘上的数据出现两个不同的页面中，通过不同的address_space对象访问。

若页面Cache中页的所有者是文件，address_space对象就嵌入在VFS inode对象中的i_data字段中。i_mapping字段总是指向含有inode数据的页所有者的address_space对象，address_space对象中的host字段指向其所有者的inode对象。

下面是VFS inode数据结构。

```
00766: struct inode {
00767:     struct hlist_node  i_hash;
00768:     struct list_head  i_list;    /* backing dev IO list */
...
00796:     struct address_space *i_mapping;
00797:     struct address_space i_data;
...
00833:     void *i_private; /* fs or device private pointer */
00834: } « end inode » ;
```

因此，若页属于一个文件（该文件存放在磁盘上文件系统中，如ext4），那么页的所有者就是文件的inode；并且对应的address_space对象存放在VFS inode对象的i_data字段中。inode的i_mapping字段指向同一个inode的i_data字段，而address_space对象的host字段也指向这个索引节点。

```
00868: if (inode_has_buffers(inode)) {
00869:     struct address_space *mapping = &inode->i_data;
00870:     struct list_head *list = &mapping->private_list;
00871:     struct address_space *buffer_mapping = mapping->assoc_mapping;
00872:
```

若页中包含的数据来自块设备文件，即页面中存放的数据是块设备的原始数据，那么就把address_space对象嵌入到与该块设备相关的特殊文件系统bdev中文件的主inode中。因此，块设备文件对应inode的i_mapping字段指向主inode中的address_space对象。相应地，address_space对象中的host字段指向主inode。这样，从块设备读取数据的所有页具有相同的address_space对象，即使这些数据位于不同的块设备文件。

我们继续分析address_space对象中的成员变量含义。

backing_dev_info字段指向struct backing_dev_info描述符，后者是对所有者的数据所在块设备进行有关描述的数据结构。backing_dev_info结构通常嵌入在块设备的请求队列描述符中。

a_ops是address_space对象中的关键字段，它指向一个类型为address_space_operations的表，表中定对所有者的页进行处理的各种方法。

2.2 address_space_operations

数据结构address_space_operations的定义在文件include/linux/fs.h中。

```
00600: struct address_space_operations {
00601:     int (*writepage)(struct page *page, struct writeback_control *wbc);
00602:     int (*readpage)(struct file *, struct page *);
00603:     void (*sync_page)(struct page *);
00604:
00605:     /* Write back some dirty pages from this mapping. */
00606:     int (*writepages)(struct address_space *, struct writeback_control *);
00607:
00608:     /* Set a page dirty. Return true if this dirtied it */
00609:     int (*set_page_dirty)(struct page *page);
00610:
00611:     int (*readpages)(struct file *filp, struct address_space *mapping,
00612:                     struct list_head *pages, unsigned nr_pages);
00613:
00614:     int (*write_begin)(struct file *, struct address_space *mapping,
00615:                       loff_t pos, unsigned len, unsigned flags,
00616:                       struct page **pagep, void **fsdata);
00617:     int (*write_end)(struct file *, struct address_space *mapping,
00618:                     loff_t pos, unsigned len, unsigned copied,
00619:                     struct page *page, void *fsdata);
00620:
00621:     /* Unfortunately this kludge is needed for FIBMAP. Don't use it */
00622:     sector_t (*bmap)(struct address_space *, sector_t);
00623:     void (*invalidatepage) (struct page *, unsigned long);
00624:     int (*releasepage) (struct page *, gfp_t);
00625:     ssize_t (*direct_IO)(int, struct kiocb *, const struct iovec *iov,
00626:                          loff_t offset, unsigned long nr_segs);
00627:     int (*get_xip_mem)(struct address_space *, pgoff_t, int,
00628:                       void **, unsigned long *);
00629:     /* migrate the contents of a page to the specified target */
00630:     int (*migratepage) (struct address_space *,
00631:                        struct page *, struct page *);
00632:     int (*launder_page) (struct page *);
00633:     int (*is_partially_uptodate) (struct page *, read_descriptor_t *,
00634:                                  unsigned long);
00635:     int (*error_remove_page)(struct address_space *, struct page *);
00636: } « end address_space_operations » ;
00637:
```

各成员变量含义如下：

writepage: 写操作，将页写到所有者所在的磁盘；
readpage: 读操作，从所有者的磁盘上读取页；
sync_page: 若对所有者拥有的页的操作准备好，则立即开始I/O数据传输；
writepages: 将所有者的多个脏页写到磁盘上；
set_page_dirty: 将所有者的页的状态，设为脏；
readpages: 从磁盘上读取多个所有者的页；
prepare_write: 准备一个写操作（由磁盘文件系统使用）；
commit_write: 完成一个写操作（由磁盘文件系统使用）；
bmap: 从文件块索引中，获取逻辑块号；
invalidatepage: 使拥有者的页无效（截断文件时使用）；
releasepage: 准备释放页，由日志文件系统使用；
direct_IO: 对所有者的页进行直接I/O数据传输（不经过页面Cache）。

上面最重要的方法是readpage/readpages、writepage/writepages、prepare_write和commit_write。在大多数情况下，这些方法把所有者的inode对象和访问物理设备的底层设备驱动联系起来。如，为普通文件的inode实现readpage方法的函数知道如何确定文件页的对应块在物理磁盘上的位置。

2.3 find_get_page ()

对页面Cache操作的基本高级函数有查找、增加和删除页。本节介绍页面的查找函数

find_get_page ()，函数在mm/filemap.c文件中。

find_get_page ()的参数有两个：address_space对象的指针和文件页面偏移量。若在基树中找到了指定页，就增加该页的计数。

```

00681: /**
00682: * find_get_page - find and get a page reference
00683: * @mapping: the address_space to search
00684: * @offset: the page index
00685: *
00686: * Is there a pagecache struct page at the given (mapping, offset) tuple?
00687: * If yes, increment its refcount and return it; if no, return NULL.
00688: */
00689: struct page *find_get_page(struct address_space *mapping, pgoff_t
                                offset)
00690: {
00691:     void **pagep;
00692:     struct page *page;
00693:
  
```

```

00694:   rcu_read_lock();
00695: repeat:
00696:   page = NULL;
00697:   pagep = radix_tree_lookup_slot(&mapping->page_tree, offset);
00698:   if (pagep) {
00699:       page = radix_tree_deref_slot(pagep);
00700:       if (unlikely(! page || page == RADIX_TREE_RETRY))
00701:           goto ↑repeat;
00702:
00703:       if (!page_cache_get_speculative(page))
00704:           goto ↑repeat;
00705:
00706:       /*
00707:        * Has the page moved?
00708:        * This is part of the lockless pagecache protocol. See
00709:        * include/linux/pagemap.h for details.
00710:        */
00711:       if (unlikely(page != *pagep)) {
00712:           page_cache_release(page);
00713:           goto ↑repeat;
00714:       }
00715:   }
00716:   rcu_read_unlock();
00717:
00718:   return page;
00719: } « end find_get_page »
00720: EXPORT_SYMBOL(find_get_page);

```

函数的主体就是radix_tree_lookup_slot()，页面Cache中的页是以基树(radix tree)的方式保存。在基树中查找某页，就使用radix_tree_lookup_slot()函数。至于基树在内核中的实现，不作介绍。

2.4 add_to_page_cache()

函数add_to_page_cache()的作用是将一个新页插入到页面Cache中，源码在文件

include/linux/pagemap.h中。

```

00469: * Like add_to_page_cache_locked, but used to add newly allocated pages:
00470: * the page is new, so we can just run ____set_page_locked() against it.
00471: */
00472: static inline int add_to_page_cache(struct page *page,
00473:                                     struct address_space *mapping, pgoff_t offset, gfp_t gfp_mask)
00474: {
00475:     int error;
00476:
00477:     ____set_page_locked(page);
00478:     error = add_to_page_cache_locked(page, mapping, offset, gfp_mask);
00479:     if (unlikely(error))
00480:         ____clear_page_locked(page);
00481:     return error;
00482: }
00483:

```

函数有4个参数：

page: 页面描述符指针，该页面中有文件数据；

mapping: `address_space`对象指针；

offset: 表示该页在文件的页面索引；

gfp_mask: 为基树分配新节点时所使用的分配标志。

将页面插入基树的主体是`radix_tree_insert()`，这里我们只要清楚是将一个页面（该页面中的数据是从磁盘读取上来的）插入到页面Cache中就可以了。

2.5 `remove_from_page_cache()`

函数`remove_from_page_cache()`作用是从页面Cache中删除某个页，源码在文件`mm/filemap.c`中。

```
00144: void remove_from_page_cache(struct page *page)
00145: {
00146:     struct address_space *mapping = page->mapping;
00147:     void (*freepage)(struct page *) = NULL;
00148:     struct inode *inode = mapping->host;
00149:
00150:     BUG_ON(! PageLocked(page));
00151:
00152:     if (IS_AOP_EXT(inode))
00153:         freepage = EXT_AOPS(mapping->a_ops)->freepage;
00154:
00155:     spin_lock_irq(&mapping->tree_lock);
00156:     __remove_from_page_cache(page);
00157:     spin_unlock_irq(&mapping->tree_lock);
00158:     mem_cgroup_uncharge_cache_page(page);
00159:
00160:     if (freepage)
00161:         freepage(page);
00162: }
00163: EXPORT_SYMBOL(remove_from_page_cache);
00164:
```

从页面Cache中删除某页主要由`radix_tree_delete()`函数来完成。若对函数`radix_tree_lookup()`、`radix_tree_insert()`和`radix_tree_delete()`感兴趣，可自行研究内核基树的实现。

3 Linux内核预读机制

大多数磁盘I/O读写都是顺序的，且普通文件在磁盘上的存储都是占用连续的扇区。这样读写文件时，就可以减少磁头的移动次数，提升读写性能。当程序读一个文件时，它通常从第一字节到最后一个字节顺序访问。因此，同一个文件中磁盘上多个相邻的扇区通常会被读进程都访问。

预读（read ahead）就是在数据真正被访问之前，从普通文件或块设备文件中读取多个连续的文件页面到内存中。多数情况下，内核的预读机制可以明显提高磁盘性能，因为减少了磁盘控制器处理的命令数，每个命令读取多个相邻扇区。此外，预读机制还提高了系统响应时间。

当然，在进程大多数的访问是随机读时，预读是对系统有害的，因为它浪费了内核Cache空间。当内核确定最近常用的I/O访问不是顺序的时，就会减少或关闭预读。

预读(read-ahead)算法预测即将访问的页面，并提前把它们批量的读入缓存。

它的主要功能和任务包括：

- 批量：把小I/O聚集为大I/O，以改善磁盘的利用率，提升系统的吞吐量。
- 提前：对应用程序隐藏磁盘的I/O延迟，以加快程序运行。
- 预测：这是预读算法的核心任务。前两个功能的达成都有赖于准确的预测能力。

当前包括Linux、FreeBSD和Solaris等主流操作系统都遵循了一个简单有效的原则：

把读模式分为随机读和顺序读两大类，并只对顺序读进行预读。这一原则相对保守，但是可以保证很高的预读命中率，同时有效率/覆盖率也很好。因为顺序读是最简单而普遍的，而随机读在内核来说也确实难以预测的。

文件预读的几种情况：

在以下情况下，执行预读：

- 当内核处理用户进程的读文件数据请求；此时调用page_cache_sync_readahead（）或page_cache_async_readahead（），我们已看到它被函数do_generic_file_read（）调用；该函数我们稍后会详细分析。
- 当内核为文件内存映射（memory mapping）分配一个页面时；
- 用户程序执行系统调用readahead（）；
- 当用户程序执行posix_fadvise（）系统调用；

当用户程序执行 `madvise()` 系统调用，使用 `MADV_WILLNEED` 命令，来通知内核文件内存映射的特定区域将来会被访问。

对文件预读需要复杂的算法：

- ◆ 读数据是按页为单位进行，不需要考虑从页面内的偏移量，仅考虑文件的访问页面部分。
- ◆ 只要进程不断顺序读数据，预读可能逐渐递增。
- ◆ 当前的访问与前面的访问不是顺序的时（即随机访问），预读必须缩小或停止。
- ◆ 当进程不断访问文件同一个页面时（仅访问文件的一小部分），或者文件的所有页面均已在页面 `cache` 中时，必须停止预读。

内核判断两次读访问是顺序的标准是：请求的第一个页面与上次访问的最后一个页面是相邻的。访问一个给定的文件，预读算法使用两个页面集：当前窗口（`current window`）和前进窗口（`ahead window`）。

当前窗口（`current window`）包括了进程已请求的页面或内核提前读且在页面 `cache` 中的页面。（当前窗口中的页面未必是最新的，因为可能仍有 I/O 数据传输正在进行。）前进窗口（`ahead window`）包含的页面是紧邻当前窗口（`current window`）中内核提前读的页面。前进窗口中的页面没有被进程请求，但内核假设进程迟早会访问这些页面。

当内核判断出一个顺序访问和初始页面属于当前窗口时，就检查前进窗口是否已经建立起来。若未建立，内核建立一个新的前进窗口，并且为对应的文件页面触发读操作。在理想状况下，进程正在访问的页面都在当前窗口中，而前进窗口中的文件页面正在传输。当进程访问的页面在前进窗口中时，前进窗口变为当前窗口。

3.1 file_ra_state 数据结构

预读算法使用的主要数据结构是 `file_ra_state`，每个文件对象都有一个 `f_ra` 域。其定义在文件 `include/linux/fs.h` 中。

```
00935: /*
00936:  * Track a single file's readahead state
00937:  */
00938: struct file_ra_state {
00939:     pgoff_t start;          /* where readahead started */
00940:     unsigned int size;      /* # of readahead pages */
00941:     unsigned int async_size; /* do asynchronous readahead when
00942:                               there are only # of pages ahead */
00943: }
```



```

00944:    unsigned int ra_pages;          /* Maximum readahead window */
00945:    unsigned int mmap_miss;         /* Cache miss stat for mmap accesses */
00946:    loff_t prev_pos;               /* Cache last read() position */
00947: };

```

表2 file_ra_state数据结构成员变量含义

字段	含义
start	当前窗口的第一个页面索引
size	当前窗口的页面数量。值为-1表示预读临时关闭，0表示当前窗口为空
async_size	异步预读页面数量
ra_pages	预读窗口最大页面数量。0表示预读暂时关闭。
mmap_miss	预读失效计数
prev_pos	Cache中最近一次读位置

ra_pages表示当前窗口的最大页面数，也就是针对该文件的最大预读页面数；其初始值由该文件所在块设备上的backing_dev_info描述符中。进程可以通过系统调用posix_fadvise（）来改变已打开文件的ra_page值来调优预读算法。

3.2 do_generic_file_read（）

现在我们回顾一下do_generic_mapping_read（）函数执行流程和

page_cache_async_readahead（）函数调用栈。

```

[<ffffff8112a790>] ? page_cache_async_readahead+0x90/0xc0
[<ffffff81115e13>] ? generic_file_aio_read+0x503/0x700
[<ffffff8117aeaa>] ? do_sync_read+0xfa/0x140
[<ffffff810920d0>] ? autoremove_wake_function+0x0/0x40
[<ffffff810edfc2>] ? ring_buffer_lock_reserve+0xa2/0x160
[<ffffff810d69e2>] ? audit_syscall_entry+0x272/0x2a0
[<ffffff81213136>] ? security_file_permission+0x16/0x20
[<ffffff8117b8b5>] ? vfs_read+0xb5/0x1a0
[<ffffff8117b9f1>] ? sys_read+0x51/0x90
[<ffffff8100b308>] ? tracesys+0xd9/0xde

```

内核处理用户进程的读数据请求时，使用最多的是调用page_cache_sync_readahead（）和page_cache_async_readahead（）函数来执行预读。

```

01038: static void do_generic_file_read(struct file *filp, loff_t *ppos,
01039:    read_descriptor_t *desc, read_actor_t actor)
01040: {
01041:    struct address_space *mapping = filp->f_mapping;
01042:    struct inode *inode = mapping->host;
01043:    struct file_ra_state *ra = &filp->f_ra;
01044:
01045:    ...
01063:    cond_resched();
01064:    find_page:

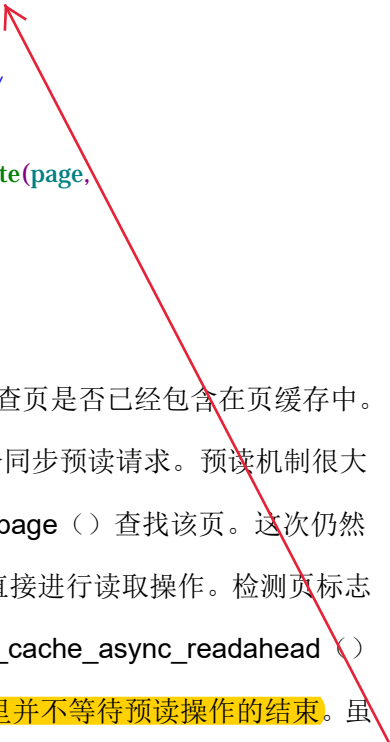
```



```

01065:     page = find_get_page(mapping, index);
01066:     if (!page) {
01067:         page_cache_sync_readahead(mapping,
01068:             ra, filp,
01069:             index, last_index - index);
01070:         page = find_get_page(mapping, index);
01071:         if (unlikely(page == NULL))
01072:             goto ↓no_cached_page;
01073:     }
01074:     if (PageReadahead(page)) {
01075:         page_cache_async_readahead(mapping,
01076:             ra, filp, page,
01077:             index, last_index - index);
01078:     }
01079:     if (!PageUptodate(page)) {
01080:         if (inode->i_blkbits == PAGE_CACHE_SHIFT ||
01081:             !mapping->a_ops->is_partially_uptodate)
01082:             goto ↓page_not_up_to_date;
01083:         if (!trylock_page(page))
01084:             goto ↓page_not_up_to_date;
01085:         /* Did it get truncated before we got the lock? */
01086:         if (!page->mapping)
01087:             goto ↓page_not_up_to_date_locked;
01088:         if (!mapping->a_ops->is_partially_uptodate(page,
01089:             desc, offset))
01090:             goto ↓page_not_up_to_date_locked;
01091:         unlock_page(page);
01092:     }
... ..

```



do_generic_file_read() 首先调用find_get_page() 检查页是否已经包含在页缓存中。如果没有则调用page_cache_sync_readahead() 发出一个同步预读请求。预读机制很大程度上能够保证数据已经进入缓存，因此再次调用find_get_page() 查找该页。这次仍然有一定失败的概率，那么就跳转到标号no_cached_page处直接进行读取操作。检测页标志是否设置了PG_readahead，如果设置了该标志就调用page_cache_async_readahead() 启动一个异步预读操作，这与前面的同步预读操作不同，这里并不等待预读操作的结束。虽然页在缓存中了，但是其数据不一定是最新的，这里通过PageUptodate(page)来检查。如果数据不是最新的，则调用函数mapping->a_ops->readpage() 进行再次数据读取。

本章节重点分析预读代码。

3.3 page_cache_sync_readahead()

我们先分析page_cache_sync_readahead() 函数，顾名思义，就是同步预读一些页面到内存中。

page_cache_sync_readahead() 它重新装满当前窗口和前进窗口，并根据预读命中率来更新窗口大小。函数有5个参数：

mapping: 文件拥有者的addresss_space对象

ra: 包含此页面的文件file_ra_state描述符

filp: 文件对象

offset: 页面在文件内的偏移量

req_size: 完成当前读操作需要的页面数

```

00482: /**
00483:  * page_cache_sync_readahead - generic file readahead
00484:  * @mapping: address_space which holds the pagecache and I/O vectors
00485:  * @ra: file_ra_state which holds the readahead state
00486:  * @filp: passed on to ->readpage() and ->readpages()
00487:  * @offset: start offset into @mapping, in pagecache page-sized units
00488:  * @req_size: hint: total size of the read which the caller is performing in
00489:  *           pagecache pages
00490:  *
00491:  * page_cache_sync_readahead() should be called when a cache miss happened:
00492:  * it will submit the read. The readahead logic may decide to piggyback more
00493:  * pages onto the read request if access patterns suggest it will improve
00494:  * performance.
00495:  */
00496: void page_cache_sync_readahead(struct address_space
00497:                                *mapping, struct file_ra_state *ra, struct file *filp,
00498:                                pgoff_t offset, unsigned long req_size)
00499: {
00500:     /* no read-ahead */
00501:     if (!ra->ra_pages)
00502:         return;
00503:
00504:     /* be dumb */
00505:     if (filp && (filp->f_mode & FMODE_RANDOM)) {
00506:         force_page_cache_readahead(mapping, filp, offset, req_size);
00507:         return;
00508:     }
00509:
00510:     /* do read-ahead */
00511:     ondemand_readahead(mapping, ra, filp, false, offset, req_size);
00512: }
00513: EXPORT_SYMBOL_GPL(page_cache_sync_readahead);
00514:

```

当文件模式设置FMODE_RANDOM时，表示文件预期为随机访问。这种情形比较少见，这里不关注。函数变成对ondemand_readahead（）封装。

3.4 page_cache_async_readahead（）

page_cache_async_readahead（）源码也在mm/readahead.c文件中，异步提前读取多个页面到内存中。

函数共6个参数，比page_cache_sync_readahead（）多一个参数page。

```

00530: void

```

```

00531: page_cache_async_readahead(struct address_space
00532:     *mapping, struct file_ra_state *ra, struct file *filp,
00533:     struct page *page, pgoff_t offset,
00534:     unsigned long req_size)
00535: {
00536:     /* no read-ahead */
00537:     if (!ra->ra_pages)
00538:         return;
00539:
00540:     /*
00541:      * Same bit is used for PG_readahead and PG_reclaim.
00542:      */
00543:     if (PageWriteback(page))
00544:         return;
00545:
00546:     ClearPageReadahead(page);
00547:
00548:     /*
00549:      * Defer asynchronous read-ahead on IO congestion.
00550:      */
00551:     if (bdi_read_congested(mapping->backing_dev_info))
00552:         return;
00553:
00554:     /* do read-ahead */
00555:     ondemand_readahead(mapping, ra, filp, true, offset, req_size);
00556:
00557:     #ifdef CONFIG_BLOCK
00558:     /*
00559:      * Normally the current page is !uptodate and lock_page() will be
00560:      * immediately called to implicitly unplug the device. However this
00561:      * is not always true for RAID configurations, where data arrives
00562:      * not strictly in their submission order. In this case we need to
00563:      * explicitly kick off the IO.
00564:      */
00565:     if (PageUptodate(page))
00566:         blk_run_backing_dev(mapping->backing_dev_info, NULL);
00567:     #endif
00568: } « end page_cache_async_readahead »
00569: EXPORT_SYMBOL_GPL(page_cache_async_readahead);

```

若不需要预读（537行）或者页面处于回写状态（543行），就直接返回。

通过前面的检查后，就清除页面PG_readahead标志（546行）。

在执行预读前，还要检查当前磁盘I/O是否处于拥塞状态，若处于拥塞就不能再进行预读。接下来就调用ondemand_readahead（）真正执行预读。

3.5 ondemand_readahead（）

ondemand_readahead（）函数实现在文件mm/readahead.c。

该函数主要根据file_ra_state描述符中的成员变量来执行一些动作。

（1）首先判断如果是从文件头开始读取的，初始化预读信息。默认设置预读为4个page。

（2）如果不是文件头，则判断是否连续的读取请求，如果是则扩大预读数量。一般等于上次预读数量x2。

(3) 否则就是随机的读取，不适用预读，只读取sys_read请求的数量。

(4) 然后调用ra_submit提交读取请求。

```

00385: /*
00386:  * A minimal readahead algorithm for trivial sequential/random reads.
00387:  */
00388: static unsigned long
00389: ondemand_readahead(struct address_space *mapping,
00390:                     struct file_ra_state *ra, struct file *filp,
00391:                     bool hit_readahead_marker, pgoff_t offset,
00392:                     unsigned long req_size)
00393: {
00394:     unsigned long max = max_sane_readahead(ra->ra_pages);
00395:
00396:     /*
00397:      * start of file
00398:      */
00399:     if (!offset)
00400:         goto initial_readahead;
00401:
00402:     /*
00403:      * It's the expected callback offset, assume sequential access.
00404:      * Ramp up sizes, and push forward the readahead window.
00405:      */
00406:     if ((offset == (ra->start + ra->size - ra->async_size) ||
00407:         offset == (ra->start + ra->size))) {
00408:         ra->start += ra->size;
00409:         ra->size = get_next_ra_size(ra, max);
00410:         ra->async_size = ra->size;
00411:         goto readit;
00412:     }
00413:
00414:     /*
00415:      * Hit a marked page without valid readahead state.
00416:      * E.g. interleaved reads.
00417:      * Query the pagecache for async_size, which normally equals to
00418:      * readahead size. Ramp it up and use it as the new readahead size.
00419:      */
00420:     if (hit_readahead_marker) {
00421:         pgoff_t start;
00422:
00423:         rcu_read_lock();
00424:         start = radix_tree_next_hole(&mapping->page_tree, offset+1,max);
00425:         rcu_read_unlock();
00426:
00427:         if (!start || start - offset > max)
00428:             return 0;
00429:
00430:         ra->start = start;
00431:         ra->size = start - offset; /* old async_size */
00432:         ra->size += req_size;
00433:         ra->size = get_next_ra_size(ra, max);
00434:         ra->async_size = ra->size;
00435:         goto readit;
00436:     }
00437:
00438:     /*
00439:      * oversize read
00440:      */
00441:     if (req_size > max)

```

req_size: 进行读操作一共需要读取页面的个数 = last_index - index

```

00442:         goto ↓initial_readahead;
00443:
00444:     /*
00445:     * sequential cache miss
00446:     */
00447:     if (offset - (ra->prev_pos >> PAGE_CACHE_SHIFT) <= 1UL)
00448:         goto ↓initial_readahead;
00449:
00450:     /*
00451:     * Query the page cache and look for the traces(cached history pages)
00452:     * that a sequential stream would leave behind.
00453:     */
00454:     if (try_context_readahead(mapping, ra, offset, req_size, max))
00455:         goto ↓readit;
00456:
00457:     /*
00458:     * standalone, small random read
00459:     * Read as is, and do not pollute the readahead state.
00460:     */
00461:     return __do_page_cache_readahead(mapping, filp, offset, req_size, 0);
00462:
00463: initial_readahead:
00464:     ra->start = offset;
00465:     ra->size = get_init_ra_size(req_size, max);
00466:     ra->async_size = ra->size > req_size ? ra->size - req_size : ra->size;
00467:
00468: readit:
00469:     /*
00470:     * Will this read hit the readahead marker made by itself?
00471:     * If so, trigger the readahead marker hit now, and merge
00472:     * the resulted next readahead window into the current one.
00473:     */
00474:     if (offset == ra->start && ra->size == ra->async_size) {
00475:         ra->async_size = get_next_ra_size(ra, max);
00476:         ra->size += ra->async_size;
00477:     }
00478:
00479:     return ra_submit(ra, mapping, filp);
00480: } « end ondemand_readahead »
00481:

```

get_init_ra_size () 计算初始预读窗口大小, get_next_ra_size () 计算下一个预读窗口大小。

当进程第一次访问文件并且请求的第一个页面在文件内的偏移量是0时, ondemand_readahead () 函数会认为进程会进行顺序访问文件。于是函数从第一个页面开始创建新的当前窗口。当前窗口的初始值大小一般是2的幂次方, 通常与进程第一次读取的页面数有关: 请求的页面数越多, 当前窗口越大, 最大值保存在ra->ra_pages中。相反地, 进程第一次访问文件, 但请求的页面在文件内的偏移量不是0时, 内核就认为进程不会进行顺序访问。这样暂时关闭预读功能(设置ra->size的值为-1)。然而内核重新发现顺序访问文件时, 就会启用预读, 创建新的当前窗口。

若前进窗口 (ahead window) 不存在, 当函数意识到进程在当前窗口进行顺序访问时,

就会创建新的前进窗口。前进窗口的起始页面通常是紧邻当前窗口的最后一个页面。前进窗口的大小与当前窗口大小有关。

一旦函数发现文件访问不是顺序的（根据前一次的访问），当前窗口和前进窗口就会被清空且预读功能被暂时关闭。当发现顺序访问时，就会重新启用预读。

3.6 ra_submit ()

ra_submit () 仅是对 __do_page_cache_readahead () 的封装。

```
00241: /*
00242:  * Submit IO for the read-ahead request in file_ra_state.
00243:  */
00244: unsigned long ra_submit(struct file_ra_state *ra,
00245:                          struct address_space *mapping, struct file *filp)
00246: {
00247:     int actual;
00248:
00249:     actual = __do_page_cache_readahead(mapping, filp,
00250: ra->start, ra->size, ra->async_size);
00251:
00252:     return actual;
00253: }
```

3.7 __do_page_cache_readahead ()

__do_page_cache_readahead () 有4个参数：

mapping: 文件拥有者的addresss_space对象

filp: 文件对象

offset: 页面在文件内的偏移量

nr_to_read: 完成当前读操作需要的页面数

lookahead_size: 异步预读大小

```
00135: /*
00136:  * ____do_page_cache_readahead() actually reads a chunk of disk. It allocates all
00137:  * the pages first, then submits them all for I/O. This avoids the very bad
00138:  * behaviour which would occur if page allocations are causing VM writeback.
00139:  * We really don't want to intermingle reads and writes like that.
00140:  *
00141:  * Returns the number of pages requested, or the maximum amount of I/O allowed.
00142:  */
00143: static int
00144: __do_page_cache_readahead(struct address_space
                          *mapping, struct file *filp,
                          pgoff_t offset, unsigned long nr_to_read,
                          unsigned long lookahead_size)
```

```

00147: {
00148:     struct inode *inode = mapping->host;
00149:     struct page *page;
00150:     unsigned long end_index; /* The last page we want to read */
00151:     LIST_HEAD(page_pool);
00152:     int page_idx;
00153:     int ret = 0;
00154:     loff_t isize = i_size_read(inode);
00155:
00156:     if (isize == 0)
00157:         goto out;
00158:
00159:     end_index = ((isize - 1) >> PAGE_CACHE_SHIFT);
00160:
00161:     /*
00162:     * Preallocate as many pages as we will need.
00163:     */
00164:     for (page_idx = 0; page_idx < nr_to_read; page_idx++) {
00165:         pgoff_t page_offset = offset + page_idx;
00166:
00167:         if (page_offset > end_index)
00168:             break;
00169:
00170:         rcu_read_lock();
00171:         page = radix_tree_lookup(&mapping->page_tree, page_offset);
00172:         rcu_read_unlock();
00173:         if (page)
00174:             continue;
00175:
00176:         page = page_cache_alloc_cold(mapping);
00177:         if (!page)
00178:             break;
00179:         page->index = page_offset;
00180:         list_add(&page->lru, &page_pool);
00181:         if (page_idx == nr_to_read - lookahead_size)
00182:             SetPageReadahead(page);
00183:         ret++;
00184:     } << end for page_idx=0;page_idx<n... >>
00185:

```



164~184行在从磁盘上读数据前，首先预分配一些内存页面，用来存放读取的文件数据。在预读过程中，可能有其他进程已经将某些页面读进内存，检因此在此检查页面是否已经在Cache中（170~172行）。若页面Cache中没有所请求的页面，则分配内存页面（176行），并将页面加入到页面池中（180行）。

当分配到第nr_to_read - lookahead_size个页面时，就设置该页面标志PG_readahead，以让下次进行异步预读（181~182行）。

```

00186:     /*
00187:     * Now start the IO. We ignore I/O errors - if the page is not
00188:     * uptodate then the caller will launch readpage again, and
00189:     * will then handle the error.
00190:     */
00191:     if (ret)
00192:         read_pages(mapping, filp, &page_pool, ret);
00193:     BUG_ON(!list_empty(&page_pool));
00194: out:
00195:     return ret;
00196: } << end ____do_page_cache_readahead >>
00197:

```

页面准备好后，调用read_pages（）执行I/O操作，从磁盘读取文件数据。

3.8 read_pages（）

read_pages（）函数源码在mm/readahead.c中。

```

00108: static int read_pages(struct address_space *mapping, struct file *filp,
00109:      struct list_head *pages, unsigned nr_pages)
00110: {
00111:     unsigned page_idx;
00112:     int ret;
00113:
00114:     if (mapping->a_ops->readpages) {
00115:         ret = mapping->a_ops->readpages(filp, mapping, pages, nr_pages);
00116:         /* Clean up the remaining pages */
00117:         put_pages_list(pages);
00118:         goto out;
00119:     }
00120:
00121:     for (page_idx = 0; page_idx < nr_pages; page_idx++) {
00122:         struct page *page = list_to_page(pages);
00123:         list_del(&page->lru);
00124:         if (!add_to_page_cache_lru(page, mapping,
00125:             page->index, GFP_KERNEL)) {
00126:             mapping->a_ops->readpage(filp, page);
00127:         }
00128:         page_cache_release(page);
00129:     }
00130:     ret = 0;
00131: out:
00132:     return ret;
00133: } « end read_pages »

```

ext4文件系统的address_space_operations对象中的readpages方法实现为

ext4_readpages（）。若readpages方法没有定义，则readpage方法来每次读取一页。从方法名字，我们很容易看出两者的区别，readpages是一次可以读取多个页面，readpage是每次只读取一个页面。两个方法实现上差别不大，我们以ext4文件系统为例，只考虑readpages方法。

ext4_readpages（）在文件fs/ext4/inode.c中。

```

03555: static int
03556: ext4_readpages(struct file *file, struct address_space *mapping,
03557:      struct list_head *pages, unsigned nr_pages)
03558: {
03559:     return mpage_readpages(mapping, pages, nr_pages, ext4_get_block);
03560: }
03561:

```

ext4_readpages（）是对mpage_readpages（）的封装。请注意get_block_t方法，ext4文件系统中对应的函数是ext4_get_block（）。

3.9 mpage_readpages ()

mpage_readpages () 的实现在fs/mpage.c中。

```

00371: int
00372: mpage_readpages(struct address_space *mapping, struct
00373:     list_head *pages, unsigned nr_pages, get_block_t get_block)
00374: {
00375:     struct bio *bio = NULL;
00376:     unsigned page_idx;
00377:     sector_t last_block_in_bio = 0;
00378:     struct buffer_head map_bh;
00379:     unsigned long first_logical_block = 0;
00380:
00381:     map_bh.b_state = 0;
00382:     map_bh.b_size = 0;
00383:     for (page_idx = 0; page_idx < nr_pages; page_idx++) {
00384:         struct page *page = list_entry(pages->prev, struct page, lru);
00385:
00386:         prefetchw(&page->flags);
00387:         list_del(&page->lru);
00388:         if (!add_to_page_cache_lru(page, mapping,
00389:             page->index, GFP_KERNEL)) {
00390:             bio = do_mpage_readpage(bio, page,
00391:                 nr_pages - page_idx,
00392:                 &last_block_in_bio, &map_bh,
00393:                 &first_logical_block,
00394:                 get_block);
00395:         }
00396:         page_cache_release(page);
00397:     }
00398:     BUG_ON(!list_empty(pages));
00399:     if (bio)
00400:         mpage_bio_submit(READ, bio);
00401:     return 0;
00402: } « end mpage_readpages »
00403: EXPORT_SYMBOL(mpage_readpages);

```

这个函数也非常简单，383行是个for循环，循环次数就是要读取的页面数。循环体中主要调用do_mpage_readpage ()，每次读取一个页面（若读请求的页面在磁盘上连续，则会do_mpage_readpage ()会将合并成一个bio请求）。从这里我们可以看出，内核事实上是以一个页面为单位从磁盘上读取数据的。在读取数据前，先将页面加入到页面Cache中（388行）。

mpage_readpages尽量少的构造bio来提交读取请求。假设要读取8个page，并且他们在磁盘上都是连续的，那么mpage_readpages仅会构造一个bio，不通的page请求保存在bio的bio_vec不同段中。当然很多时候，mpage_readpages多个page会产生多个bio。

399~400行根据前面构造的bio，提交读请求。

do_mpage_readpage（）函数及之后的调用，可参考linux内核读文件过程。

4 文件读写I/O流程与Cache机制

前面我们分析了页面Cache的由来和作用，以及相关重要的数据结构。在本节中，我们再回顾分析文件读写I/O流程与页面Cache的关系。

4.1 读文件过程

下面内核读文件函数调用栈，我们可以获取函数调用顺序。分析读文件过程和页面Cache之间的关系，就略去无关部分。内核读文件完整过程，可以参考Linux内核读文件过程。

```
Pid: 3823, comm: md5sum Tainted: G      ----- HT 2.6.32279.debug #46
Call Trace:
[<ffffffff812572ed>] ? submit_bio+0x11d/0x1b0
[<ffffffff8112b560>] ? __lru_cache_add+0x40/0x90
[<ffffffff811b6c67>] ? mpage_bio_submit+0x27/0x30
[<ffffffff811b7565>] ? mpage_readpages+0x115/0x130
[<fffffffa00e0730>] ? ext4_get_block+0x0/0x120 [ext4]
[<fffffffa00e0730>] ? ext4_get_block+0x0/0x120 [ext4]
[<ffffffff8115c1da>] ? alloc_pages_current+0xaa/0x110
[<fffffffa00dcafd>] ? ext4_readpages+0x1d/0x20 [ext4]
[<ffffffff8112a1b5>] ? __do_page_cache_readahead+0x185/0x210
[<ffffffff8112a261>] ? ra_submit+0x21/0x30
[<ffffffff8112a5d5>] ? ondemand_readahead+0x115/0x240
[<ffffffff810724c7>] ? current_fs_time+0x27/0x30
[<ffffffff8112a790>] ? page_cache_async_readahead+0x90/0xc0
[<ffffffff81115e13>] ? generic_file_aio_read+0x503/0x700
[<ffffffff8117aeaa>] ? do_sync_read+0xfa/0x140
[<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
[<ffffffff810edfc2>] ? ring_buffer_lock_reserve+0xa2/0x160
[<ffffffff810d69e2>] ? audit_syscall_entry+0x272/0x2a0
[<ffffffff81213136>] ? security_file_permission+0x16/0x20
[<ffffffff8117b8b5>] ? vfs_read+0xb5/0x1a0
[<ffffffff8117b9f1>] ? sys_read+0x51/0x90
[<ffffffff8100b308>] ? tracesys+0xd9/0xde
```

用户进程读取文件数据，有两种情形：

- (1) 所需要的数据不在内存中，也即不在页面Cache中。这时就需要直接从磁盘上读取。
- (2) 所需的数据已经在内存中，此时只需从页面Cache中找到具体位置，然后将数据拷贝到用户缓冲区。而不需要进行磁盘I/O操作。

4.1.1 数据不在页面Cache中

用户请求的数据不在页面Cache中，则表明是请求的文件数据第一次被读取（也没有被写入过）。

在不是DIRECT_IO的情况下，系统调用read()必然会执行到函数do_generic_file_read

()（文件mm/filemap.c中）下面我们就以数据不在页面Cache中来分析

do_generic_file_read () 的源码。

```

01038: static void do_generic_file_read(struct file *filp, loff_t *ppos,
01039:      read_descriptor_t *desc, read_actor_t actor)
01040: {
01041:     struct address_space *mapping = filp->f_mapping;
01042:     struct inode *inode = mapping->host;
01043:     struct file_ra_state *ra = &filp->f_ra;
01044:
01045:     ...
01057:     for (;;) {
01058:         struct page *page;
01059:         pgoff_t end_index;
01060:         loff_t isize;
01061:         unsigned long nr, ret;
01062:
01063:         cond_resched();
01064:         find_page:
01065:         page = find_get_page(mapping, index);
01066:         if (!page) {
01067:             page_cache_sync_readahead(mapping,
01068:                 ra, filp,
01069:                 index, last_index - index);
01070:             page = find_get_page(mapping, index);
01071:             if (unlikely(page == NULL))
01072:                 goto ↓no_cached_page;
01073:         }
01074:         if (PageReadahead(page)) {
01075:             page_cache_async_readahead(mapping,
01076:                 ra, filp, page,
01077:                 index, last_index - index);
01078:         }
01079:         ...

```

首先通过find_get_page () 查找请求数据是否已经在页面Cache中，在Cache找不到，就调用page_cache_sync_readahead () 将数据预读到内存中。

注意：不管用户请求的数据是否已经在页面Cache中，都会从页面Cache中查找相应的页。也就是读

文件的过程就是先从磁盘读数据到页面Cache，再从页面Cache拷贝到用户缓冲区中。

此时，不禁有人要问：初次读文件后，该部分数据何时、在哪里被加入到页面Cache中的。数据不在页面Cache中时，必然会执行函数read_pages ()，进而进入函数mpage_readpages ()。

函数 `mpage_readpages()` 的功能是读取 `nr_pages` 个页面到页面 Cache 中，并将每个页面加入到页面 Cache 中。这样以后任何一个进程请求该部分数据，就可以直接从页面 Cache 中读取了。

至此，我们分析了数据不在页面 Cache 中的情形，但 Linux 内核文件 Cache 机制，是主要提高性能，这样进程读取数据时，大都是已经在页面 Cache 中的情况，只有这样性能才会显著提升。下面我们就分析数据在页面 Cache 中的过程。

4.1.2 数据在页面 Cache 中

数据在页面 Cache 中时，表明所请求的数据已被读取到内核中。可能是该进程自己或其他进程读过这部分数据；也有可能是某个进程（包括自己）刚写过文件该部分内容；不管怎样，请求的数据已经在内核中，不需要从磁盘上读取了。本节就是分析内核如何知道所请求的数据是否在页面 Cache 中，并将数据拷贝到用户缓冲区中。

不管数据是否在内核文件 Cache 中，`read()` 系统调用都是要执行 `do_generic_mapping_read()` 函数的，我们再次分析该函数。

```

01038: static void do_generic_file_read(struct file *filp, loff_t *ppos,
01039:      read_descriptor_t *desc, read_actor_t actor)
01040: {
01041:     struct address_space *mapping = filp->f_mapping;
01042:     struct inode *inode = mapping->host;
01043:     struct file_ra_state *ra = &filp->f_ra;
01044:     ...
01057:     for (;;) {
01058:         struct page *page;
01059:         pgoff_t end_index;
01060:         loff_t isize;
01061:         unsigned long nr, ret;
01062:
01063:         cond_resched();
01064:         find_page:
01065:         page = find_get_page(mapping, index);
01066:         if (!page) {
01067:             page_cache_sync_readahead(mapping,
01068:                 ra, filp,
01069:                 index, last_index - index);
01070:             page = find_get_page(mapping, index);
01071:             if (unlikely(page == NULL))
01072:                 goto ↓no_cached_page;
01073:         }
01074:         if (PageReadahead(page)) {
01075:             page_cache_async_readahead(mapping,
01076:                 ra, filp, page,
01077:                 index, last_index - index);
01078:         }
01079:         ...

```

```

01093: page_ok:
...
01146:     ret = actor(desc, page, offset, nr);
01147:     offset += ret;
01148:     index += offset >> PAGE_CACHE_SHIFT;
01149:     offset &= ~PAGE_CACHE_MASK;
01150:     prev_offset = offset;
01151:
01152:     page_cache_release(page);
01153:     if (ret == nr && desc->count)
01154:         continue;
01155:     goto ↓out;
...
01246: out:
01247:     ra->prev_pos = prev_index;
01248:     ra->prev_pos <<= PAGE_CACHE_SHIFT;
01249:     ra->prev_pos |= prev_offset;
01250:
01251:     *ppos = ((loff_t)index << PAGE_CACHE_SHIFT) + offset;
01252:     file_accessed(filp);
01253: } « end do_generic_file_read »
01254:

```

当读请求的页面已经在页面Cache中时，1065行的find_get_page（）就会找到该页面。若数据是最新的，且该页面没有设置PG_readahead标志（和本次读无关，是上次读设置了该页标志），那么就会一直执行page_ok标号，将数据从内核拷贝到用户缓冲区，然后跳转到ou标号，结束本次读。

4.2 写文件过程

本节不分析内核写文件整个过程，相关分析可以参考章节Linux内核写文件过程。这里只介绍写文件过程中与页面Cache有关系。

这里我们还是先看一下内核写文件过程中，部分栈信息

```

[<fffffffa00e4ac8>] ? ext4_da_write_begin+0x188/0x200 [ext4]
[<fffffffa00ab63f>] ? jbd2_journal_dirty_metadata+0xff/0x150 [jbd2]
[<ffffff814ff6f6>] ? down_read+0x16/0x30
[<ffffff81114ab3>] ? generic_file_buffered_write+0x123/0x2e0
[<ffffff810724c7>] ? current_fs_time+0x27/0x30
[<ffffff81116450>] ? __generic_file_aio_write+0x250/0x480
[<ffffff8113f1c7>] ? handle_pte_fault+0xf7/0xb50
[<ffffff811166ef>] ? generic_file_aio_write+0x6f/0xe0
[<fffffffa00d9131>] ? ext4_file_write+0x61/0x1e0 [ext4]
[<ffffff8117ad6a>] ? do_sync_write+0xfa/0x140
[<ffffff810920d0>] ? autoremove_wake_function+0x0/0x40
[<ffffff810edfc2>] ? ring_buffer_lock_reserve+0xa2/0x160
[<ffffff810d69e2>] ? audit_syscall_entry+0x272/0x2a0

```

```
[<ffffff81213136>] ? security_file_permission+0x16/0x20
[<ffffff8117b068>] ? vfs_write+0xb8/0x1a0
[<ffffff8117ba81>] ? sys_write+0x51/0x90
[<ffffff8100b308>] ? tracesys+0xd9/0xde
```

在Linux内核写文件过程章节中，我们看到写入数据，必然执行generic_perform_write（）。

```
02302: static ssize_t generic_perform_write(struct file *file,
02303:                                     struct iov_iter *i, loff_t pos)
02304: {
02305:     struct address_space *mapping = file->f_mapping;
02306:     const struct address_space_operations *a_ops =
        mapping->a_ops;

    ... ..

02346:     status = a_ops->write_begin(file, mapping, pos, bytes, flags,
02347:                               &page, &fsdata);
02348:     if (unlikely(status))
02349:         break;
02350:
    ... ..
```

函数执行过程中，会调用，a_ops->write_begin（）方法。这里以ext4文件系统deley allocation模式为例，看代码实现。

```
03265: static int ext4_da_write_begin(struct file *file, struct
03265:                                address_space * mapping,
03266:                                loff_t pos, unsigned len, unsigned flags,
03267:                                struct page **pagep, void **fsdata)
03268: {
03269:     int ret, retries = 0;
03270:     struct page *page;
03271:     pgoff_t index;
03272:     unsigned from, to;
03273:     struct inode *inode = mapping->host;
03274:     handle_t *handle;

    ... ..

03303:     page = grab_cache_page_write_begin(mapping, index,
        flags);
03304:     if (!page) {
03305:         ext4_journal_stop(handle);
03306:         ret = -ENOMEM;
03307:         goto ↓out;

    ... ..

03328: out:
03329:     return ret;
```

03330: } ?

3303行调用`grab_cache_page_write_begin()`在`pagecache`中查找页面。如果找到了该页，则增加计数并设置`PG_locked`标志。如果该页不在页面`Cache`中，则分配一个新页，并调用`add_to_page_cache_lru()`，将该页插入页面`Cache`中，这个函数也会增加页面引用计数，并设置`PG_locked`标志。若`grab_cache_page_write_begin()`没能成功返回页面，说明系统没有空闲内存了，就没法继续写数据到硬盘。

将写入的数据页面加入`Cache`后，当有进程需要读取这个页面数据时（数据没有再次被修改），就可以直接从内存`Cache`里拷贝，不需要从硬盘里读取。