页缓存和块缓存

# 页缓存

## block\_read\_full\_page

1 int block\_read\_full\_page(struct page \*page, get\_block\_t \*get\_block)

2 {

3 struct inode \*inode = page->mapping->host;

4 sector\_t iblock, lblock;

5 struct buffer\_head \*bh, \*head, \*arr[MAX\_BUF\_PER\_PAGE];

6 unsigned int blocksize;

7 int nr, i;

8 int fully\_mapped = 1;

9

10 BUG\_ON(!PageLocked(page));

11 blocksize = 1 << inode->i\_blkbits;

12 if (!page\_has\_buffers(page))

13 //如果块缓存不存在就创建

14 create\_empty\_buffers(page, blocksize, 0);

15 //获取块缓存链表头

16 head = page\_buffers(page);

17

18 //块起始位置

19 iblock = (sector\_t)page->index << (PAGE\_CACHE\_SHIFT - inode->i\_blkbits);

20 //文件末尾的块

21 lblock = (i\_size\_read(inode)+blocksize-1) >> inode->i\_blkbits;

22 bh = head;

23 nr = 0;

24 i = 0;

25

26 /\*

27 \* 需要处理3中情况：

28 \* 1.缓冲区的内容是最新的

29 \* 2.缓冲区的内容不是最新的，有映射

30 \* 3.缓冲区的内容没有映射

31 \*/

32 do {

33 if (buffer\_uptodate(bh))

34 //如果记录块内容是一致的，就跳过

35 continue;

36

37 if (!buffer\_mapped(bh)) {

38 //如果块缓冲区还没有和物理块建立映射关系且块起始地址未超出文件末尾,建立映射

39 int err = 0;

40

41 fully\_mapped = 0;

42 if (iblock < lblock) {

43 WARN\_ON(bh->b\_size != blocksize);

44 //inode用于获取bdev, iblock指定了block number, bh是一个块缓存

45 //get\_block用于定位一个块, 即建立映射

46 err = get\_block(inode, iblock, bh, 0);

47 if (err)

48 SetPageError(page);

49 }

50 /\*

51 \*调用get\_block后，bh就会置位BH\_Mapped,如果bh没有置位BH\_Mapped,

52 \*就说明当前读的位置超过文件尾了，这时需要将内存block填充0，并设置为

53 \*BH\_Uptodate状态。

54 \*/

55 if (!buffer\_mapped(bh)) {

56 void \*kaddr = kmap\_atomic(page, KM\_USER0);

57 memset(kaddr + i \* blocksize, 0, blocksize);

58 flush\_dcache\_page(page);

59 kunmap\_atomic(kaddr, KM\_USER0);

60 if (!err)

61 set\_buffer\_uptodate(bh);

62 continue;

63 }

64 /\*

65 \* get\_block() might have updated the buffer

66 \* synchronously

67 \*/

68 /\*

69 \*某些fs会在映射期间读出数据块，将buffer设置成uptodate,

70 \*以这里需要再次检查buffer是否uptodate

71 \*/

72 if (buffer\_uptodate(bh))

73 continue;

74 }

75 //不是最新的，但是有映射的记录块的buffer\_head放到指针数组arr[]中

76 arr[nr++] = bh;

77 } while (i++, iblock++, (bh = bh->b\_this\_page) != head);

78

79 if (fully\_mapped)

80 SetPageMappedToDisk(page);

81

82 if (!nr) {

83 /\*

84 \* All buffers are uptodate - we can set the page uptodate

85 \* as well. But not if get\_block() returned an error.

86 \*/

87 if (!PageError(page))

88 SetPageUptodate(page);

89 //如果所有的在buffer head都是BH\_Uptodate状态，唤醒等待此页的进程

90 unlock\_page(page);

91 return 0;

92 }

93

94 /\* Stage two: lock the buffers \*/

95 //如果某些块处于非Uptodate状态，那么需要锁住这些block,置位BH\_Async\_Read

96 for (i = 0; i < nr; i++) {

97 bh = arr[i];

98 lock\_buffer(bh);

99 mark\_buffer\_async\_read(bh);

100 }

101

102 /\*

103 \* Stage 3: start the IO. Check for uptodateness

104 \* inside the buffer lock in case another process reading

105 \* the underlying blockdev brought it uptodate (the sct fix).

106 \*/

107 for (i = 0; i < nr; i++) {

108 bh = arr[i];

109 if (buffer\_uptodate(bh))

110 //如果buffer中的内容和disk中的一致，就从buffer中读取

111 end\_buffer\_async\_read(bh, 1);

112 else

113 submit\_bh(READ, bh);

114 }

115 return 0;

116 }

这个函数读页面的逻辑是极其清晰的：

1. 搜集判断该页面中有多少个block处于非BH\_Uptodate状态，只需判断buffer\_head的标志位即可，对于非BH\_Uptodate状态的block还需要判断该block是否已映射，即该逻辑块是否已经与物理磁盘块建立了映射关系，如果没有，那么说明可能别的进程解除了该block的映射关系（谁？为什么会解除一个页面的某些block的映射关系？），此时需要作一个简单判断，如果当前读的位置并没有超过文件末尾，那么，建立映射，否则，将内存block填充0，并设置其为BH\_Uptodate状态，至于为什么会读超过文件末尾的位置，我想这是因为，可能有别的进程在本程序读之前删除了文件的部分数据，导致文件变小，而读进程并未感知这种变化。映射完成以后，还需要判断该block是否处于BH\_Uptodate状态，这是因为某些文件系统可能在映射期间读出块数据。
2. 检查完成页面中的所有块状态以后，如果某些块处于非BH\_Uptodate状态，那么需要锁住这些block（lock\_buffer()），并设置读完成以后的回调函数mark\_buffer\_async\_read().
3. 一切就绪后，向底层IO子系统提交每一个待读出的buffer\_head。

在上面的步骤2中，首先锁定每个非BH\_Uptodate的buffer\_head，即对每个buffer\_head设置一个BH\_Locked标志位，此时其他进程无法再操作该buffer\_head对应的块直到其被解锁。锁定以后，调用函数mark\_buffer\_async\_read()来设置读完成以后的回调函数。

具体流程如图



### create\_empty\_buffers

1 void create\_empty\_buffers(struct page \*page,

2 unsigned long blocksize, unsigned long b\_state)

3 {

4 struct buffer\_head \*bh, \*head, \*tail;

5

6 //head是单向链表最头部成员

7 head = alloc\_page\_buffers(page, blocksize, 1);

8 bh = head;

9 do {

10 //将链表各成员设置b\_state

11 bh->b\_state |= b\_state;

12 tail = bh;

13 bh = bh->b\_this\_page;

14 } while (bh);

15 //尾部成员的next指向头部，完成循环链表的连接

16 tail->b\_this\_page = head;

17

18 spin\_lock(&page->mapping->private\_lock);

19 if (PageUptodate(page) || PageDirty(page)) {

20 //页中包含有效数据或者page被标记为dirty,需要将buffer\_head设置成同样的状态

21 bh = head;

22 do {

23 if (PageDirty(page))

24 //将bh->b\_state设置成BH\_Dirty

25 set\_buffer\_dirty(bh);

26 if (PageUptodate(page))

27 //将bh->b\_state设置成BH\_Uptodate

28 set\_buffer\_uptodate(bh);

29 bh = bh->b\_this\_page;

30 } while (bh != head);

31 }

32 attach\_page\_buffers(page, head);

33 spin\_unlock(&page->mapping->private\_lock);

34 }

这个函数完成后，就完成了page和buffer head的映射关系。如图。



并且page->private指向了与其相关的第一个buffer head。从19行开始，根据页的状态，设置与其关联的buffer head状态。例如一个page的标志是PG\_uptodate状态，相关的buffer head的状态就必须是BH\_Uptodate。

### alloc\_page\_buffers

1 struct buffer\_head \*alloc\_page\_buffers(struct page \*page, unsigned long size,

2 int retry)

3 {

4 struct buffer\_head \*bh, \*head;

5 long offset;

6

7 try\_again:

8 head = NULL;

9 offset = PAGE\_SIZE;

10 while ((offset -= size) >= 0) {

11 bh = alloc\_buffer\_head(GFP\_NOFS);

12 if (!bh)

13 goto no\_grow;

14

15 bh->b\_bdev = NULL;

16 bh->b\_this\_page = head;

17 bh->b\_blocknr = -1;

18 head = bh;

19

20 bh->b\_state = 0;

21 atomic\_set(&bh->b\_count, 0);

22 bh->b\_private = NULL;

23 bh->b\_size = size;

24

25 /\* Link the buffer to its page \*/

26 set\_bh\_page(bh, page, offset);

27

28 init\_buffer(bh, NULL, NULL);

29 }

30 return head;

31 /\*

32 \* In case anything failed, we just free everything we got.

33 \*/

34 no\_grow:

35 //依次释放bh

36 if (head) {

37 do {

38 bh = head;

39 head = head->b\_this\_page;

40 free\_buffer\_head(bh);

41 } while (head);

42 }

43

44 /\*

45 \* Return failure for non-async IO requests. Async IO requests

46 \* are not allowed to fail, so we have to wait until buffer heads

47 \* become available. But we don't want tasks sleeping with

48 \* partially complete buffers, so all were released above.

49 \*/

50 /\*

51 \*同步io request不经过调度算法直接操作磁盘，可能会失败。异步io request 不能失败，

52 \*所以需要等待直到获取可用的bufferhead.我们不希望获取一个获取部分可用bh的task

53 \*睡眠，因为这样会浪费一部分内存空间。所以只要bh没有获取完全，就要将已经获得的bh释放

54 \*/

55 if (!retry)

56 //如果retry为0，bh分配失败就退出

57 return NULL;

58

59 /\* We're \_really\_ low on memory. Now we just

60 \* wait for old buffer heads to become free due to

61 \* finishing IO. Since this is an async request and

62 \* the reserve list is empty, we're sure there are

63 \* async buffer heads in use.

64 \*/

65 //如果retry为1（当async io request时），如果分配失败，需要继续分配bh

66 free\_more\_memory();

67 goto try\_again;

68 }

这个函数为buffer\_head分配页缓存，假设page大小是4096byte, block size是1024byte. 这个函数建立的buffer\_head和page的关系如图



这里有点需要注意的是调用alloc\_page\_buffers函数的可能是sync io request,也可能是async io request。sync io request如果请求分配buffer head失败，就立即返回。在async io request请求分配buffer head的过程中，如果请求的buffer head只有部分分配成功，就需要释放已经分配成功的buffer head，继续请求分配buffer head，直到所有请求的buffer head全部分配成功。这是因为如果这些部分申请成功的buffer head没有释放掉，在内存吃紧的情况下，其他等待获取buffer head的async io request可能永远不能成功返回。

# 块缓存