元数据 struct buffer\_head

#define buffer\_heads\_over\_limit 0

#define PAGEVEC\_SIZE 15

enum bh\_state\_bits {

BH\_Uptodate, /\*contains valid data\*/

BH\_Dirty, /\*Is Dirty\*/

BH\_Lock, /\*Is locked\*/

BH\_Req,/\*has been submitted for I/O\*/

BH\_Boundary,/\*Block is followed by a discontiguity\*/

BH\_PrivateStart,

};

struct pagevec {

unsigned char nr;

struct page \*pages[PAGEVEC\_SIZE];

};

enum pageflags {

PG\_writeback,

};

struct buffer\_head {

/\*

circular list of page’s buffers

\*/

struct buffer\_head \*b\_this\_page;

struct page \*b\_page;/\*当前struct buffer\_head 对应的page\*/

char \*b\_data ;/\*当前struct buffer\_head所指向的page的物理地址的页内偏移\*/

};

struct page {

unsigned long private;/\*用来存储struct buffer\_head指针\*/

};

struct bio\_vec {

struct page \*bv\_page;

unsigned int bv\_len;

unsigned int bv\_offset;/\*相对于bv\_offset的\*/

};

struct bvec\_iter {

sector\_t bi\_sector;/\*记录着该bio的首个sector地址\*/

unsigned int bi\_size;/\*记录着整个bio的大小\*/

};

struct bio {

struct bvec\_iter bi\_iter;

struct bio\_vec \*bi\_io\_vec;/\*每个bio\_vec记录着连续物理地址的一系列bio

\*/

};

enum req\_opf {

REQ\_OP\_READ =0,

REQ\_OP\_WRITE =1,

REQ\_OP\_FLUSH =2,

REQ\_OP\_LAST

};

typedef int (get\_block\_t)(struct inode \*inode,sector\_t iblock,struct buffer\_head \*bh\_result,int create)

## buffer\_boundary

普通数据

在struct address\_space\_operations 的.writepages 回调回写页。

以fat32为例

fat\_writepages(struct address\_space \*mapping,struct writeback\_control \*wbc)->

mpage\_writepages(mapping,wbc,fat\_get\_block)

{

struct blk\_plug plug;

int ret;

blk\_start\_plug(&plug);

struct mpage\_data mpd = {

.bio = NULL,

.last\_block\_in\_bio =0,/\*上一个block\*/

.get\_block = get\_block,

.use\_writepage =1,

};

ret = write\_cache\_pages(mapping,wbc,\_\_mpage\_writepage,&mpd);

}

write\_cache\_pages(mapping,wbc,\_\_mpage\_writepage,&mpd)

{

struct pagevec pvec;

pgoff\_t uninitialized\_var(writeback\_index);

int range\_whole = 0;

xa\_mark\_t tag;

int done =0;

int nr\_pages;

pagevec\_init(&pvec);

/\*

如果range\_cyclic为1，则文件偏移是根据mapping->writeback\_index决定。

\*/

if(wbc->range\_cyclic) {

writeback\_index = mapping->writeback\_index;

index = writeback\_index;

end = -1;

}else {

index = wbc->range\_start >> PAGE\_SHIFT;

end = wbc->range\_end >> PAGE\_SHIFT;

if(wbc->range\_start ==0&&wbc->range\_end == LLONG\_MAX)

range\_whole =1;

}

if(wbc->sync\_mode ==WB\_SYNC\_ALL || wbc->tagged\_writepages)

tag = PAGECACHE\_TAG\_TOWRITE；

else

tag = PAGECACHE\_TAG\_DIRTY;

/\*

\*/

if(wbc->sync\_mode == WB\_SYNC\_ALL || wbc->tagged\_writepages)

tag\_pages\_for\_writeback(mapping,index,end);

while(!done &&(index <=end)) {

int i;

nr\_pages = pagevec\_lookup\_range\_tag(&pvec,mapping,&index,end,tag);

if(nr\_pages == 0)

break;

for(i=0;i<nr\_pages;i++) {

}

}

}

\_\_mpage\_writepage(struct page \*page,struct writeback\_control \*wbc,void \*data)

{

sector\_t block\_in\_file;

struct buffer\_head map\_bh;

loff\_t i\_size = i\_size\_read(inode);

unsigned page\_block;

const unsigned blkbits = inode->i\_blkbits;

const unsigned blocks\_per\_page = PAGE\_SIZE >> blkbits;

sector\_t blocks[MAX\_BUF\_PER\_PAGE];

unsigned first\_unmapped =blocks\_per\_page;

struct bio \*bio = mpd->bio;

int op\_flags = wbc\_to\_write\_flags(wbc);

struct block\_device \*bdev = NULL;

int boundary = 0;

last\_block = (i\_size -1) >> blkbits;

map\_bh.b\_page = page;

block\_in\_file = (sector\_t)page->index <<(PAGE\_SHIFT - blkbits);

for(page\_block =0;page\_block<blocks\_per\_page;) {

map\_bh.b\_state =0;

map\_bh.b\_size = 1<<blkbits;

if(mpd->get\_block(inode,block\_in\_file,&map\_bh,1))

goto confused;

if(buffer\_new(&map\_bh))

clean\_bdev\_bh\_alias(&map\_bh);

if(buffe r\_boundary(&map\_bh)) {

boundary\_block = map\_bh.b\_blocknr;

boundary\_bdev = map\_bh.b\_bdev;

}

if(page\_block) {

if(map\_bh.b\_blocknr != blocks[page\_block-1] +1)

goto confused;

}

boundary = buffer\_boundary(&map\_bh);

/\*

blocks数组记录着一个page的四个block每个block对应的物理block\_nr

\*/

blocks[page\_block++] = map\_bh.b\_blocknr;

bdev = map\_bh.b\_bdev;

if(block\_in\_file == last\_block)

break;

block\_in\_file++;

}

first\_unmapped = page\_block;

page\_is\_mapped:

end\_index = i\_size >> PAGE\_SHIFT;

if(page->index >=end\_index) {

unsigned offset = i\_size &(PAGE\_SIZE-1);

zero\_user\_segment(page,offset,PAGE\_SIZE);

}

if(bio &&mpd->last\_block\_in\_bio !=blocks[0]-1)

bio = mpage\_bio\_submit(REQ\_OP\_WRITE,op\_flags,bio);

allow\_new:

if(bio ==NULL) {

/\*

first\_unmapped记录着一个page的第一个没有unmapped的block

\*/

if(first\_unmapped ==blocks\_per\_page) {

}

bio = mpage\_alloc(bdev,blocks[0]<<(blkbits-9),BIO\_MAX\_PAGES,GFP\_NOFS |\_\_GFP\_HIGH);

}

length = first\_unmapped <<blkbits;

/\*

block的连续性已经由前面的逻辑保证了。

bio\_add\_page只有一种可能性会返回小于length，那就是当前bio已经满了。

\*/

if(bio\_add\_page(bio,page,length,0)<length) {

bio = mpage\_bio\_submit();

goto alloc\_new;

}

clean\_buffers(page,first\_unmapped);

/\*

设置page的PG\_writeback

\*/

set\_page\_writeback(page);

/\*

unlocks the page and wakes up sleepers in \_\_\_\_wait\_on\_page\_locked();

\*/

unlock\_page(page);

if(boundary || (first\_unmapped !=blocks\_per\_page)) {

bio = mpage\_bio\_submit(REQ\_OP\_WRITE,op\_flags,bio);

if(boundary\_block) {

write\_boundary\_block(boundary\_bdev,boundary\_block,1<<blkbits);

};

}else {

mpd->last\_block\_in\_bio =blocks[blocks\_per\_page-1];

}

goto out;

confused:

if(bio)

bio = mpage\_bio\_submit(REQ\_OP\_WRITE,op\_flags,bio);

if(mpd->use\_writepage) {

ret = mapping->a\_ops->writepage(page,wbc);

}

out:

mpd->bio = bio;

return ret;

}

mpage\_alloc(bdev,blocks[0]<<(blkbits-9),BIO\_MAX\_PAGES,GFP\_NOFS |\_\_GFP\_HIGH)

{

struct bio \*bio;

gfp\_flags &= GFP\_KERNEL;

bio = bio\_alloc(gfp\_flags,nr\_vecs);

if(bio) {

bio\_set\_dev(bio,bdev);

bio->bi\_iter.bi\_sector = first\_vector;

}

return bio;

}

bio\_add\_page(bio,page,length,0)

{

bool same\_page = false;

if(!\_\_bio\_try\_merge\_page(bio,page,len,offset,&same\_page))

{

if(bio\_full(bio,len))

return 0;

\_\_bio\_add\_page(bio,page,len,offset);

}

}

/\*

\_\_bio\_try\_merge\_page，如果上一个的物理地址和当前物理地址是连续的，则直接增加bv\_len和bi\_iter即可，\_\_bio\_try\_merge\_page返回true，否则，\_\_bio\_try\_merge\_page返回false。

\*/

\_\_bio\_try\_merge\_page(bio,page,len,offset,&same\_page)

{

if(bio->bi\_vcnt > 0) {

struct bio\_vec \*bv = &bio->bi\_io\_vec[bio->bi\_vcnt-1];

if(page\_is\_mergeable(bv,page,len,off,same\_page)) {

bv->bv\_len +=len;

bio->bi\_iter.bi\_size +=len;

return true;

}

}

return false;

}

page\_is\_mergeable(bv,page,len,off,same\_page)

{

phys\_addr\_t vec\_end\_addr = page\_to\_phys(bv->bv\_page) +

bv->bv\_offset + bv->bv\_len -1;

phys\_addr\_t page\_addr = page\_to\_phys(page);

if(vec\_end\_addr +1 !=page\_addr+off)

return false;

\*same\_page = ((vec\_end\_addr &PAGE\_MASK)==page\_addr);

if(!\*same\_page &&pfn\_to\_page(PFN\_DOWN(vec\_end\_addr))+1!=page)

return false;

return true;

}

\_\_bio\_add\_page(bio,page,len,offset)

{

struct bio\_vec \*bv = &bio->bi\_io\_vec[bio->bi\_vcnt];

bv->bv\_page = page;

bv->bv\_offset = off;

bv->bv\_len = len;

bio->bi\_iter.bi\_size +=len;

bio->bi\_vcnt++;

}

ret = mapping->a\_ops->writepage(page,wbc)->

fat\_writepage(page,wbc)->

block\_write\_full\_page(page,fat\_get\_block,wbc)

{

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

loff\_t i\_size = i\_size\_read(inode);

const pgoff\_t end\_index = i\_size >> PAGE\_SHIFT;

if(page->index < end\_index)

return \_\_block\_write\_full\_page(inode,page,get\_block,wbc,end\_buffer\_async\_write);

}

\_\_block\_write\_full\_page(inode,page,get\_block,wbc,end\_buffer\_async\_write)

{

int err;

sector\_t block;

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

unsigned int blocksize,bbits;

int nr\_underway = 0;

head = create\_page\_buffers(page,inode,(1<<BH\_Dirty) | (1 << BH\_Uptodate));

bh = head;

blocksize = bh->b\_size;

bbits = block\_size\_bits(blocksize);

block = (sector\_t)page->index <<(PAGE\_SHIFT -bbits);

last\_block = (i\_size\_read(inode)-1)>>bbits;

/\*

遍历struct buffer\_head

\*/

do {

if(block > last\_block) {

} else if((!buffer\_mapped(bh) || buffer\_delay(bh))&&buffer\_dirty(bh)) {

err = get\_block(inode,block,bh,1);

clear\_buffer\_delay(bh);

}

bh = bh->b\_this\_page;

block++;

}while(bh!=head)

do {

if(!buffer\_mapped(bh))

continue;

if(wbc->sync\_mode !=WB\_SYNC\_NONE) {

/\*

lock\_buffer，首先调用trylock\_buffer,置位BH\_Lock,并返回BH\_Lock的原来的值。如果原来的值是0，则直接返回，如果原来的值是1，则则调用\_\_lock\_buffer,\_\_lock\_buffer，首先置位struct buffer\_head的BH\_Lock,并返回struct buffer\_head的BH\_Lock的原来值，如果原来值是0，则直接返回0，如果原来值是1，则进入wait\_on\_bit等待阶段。

lock\_buffer的总逻辑是，如果原来的值是1，则进入等待；如果原来的值是0，则直接置位并返回。

\*/

lock\_buffer(bh);

}else if(!trylock\_buffer(bh)) {

}

/\*

test\_clear\_buffer\_dirty，顾名思义，先清楚BH\_Dirty，再返回BH\_Dirty原来的值

\*/

if(test\_clear\_buffer\_dirty(bh)) {

/\*

初始化b\_end\_io，并设置bh的BH\_Async\_Write

\*/

mark\_buffer\_async\_write\_endio(bh,handler);

}else {

unlock\_buffer(bh);

}

}while((bh=bh->b\_this\_page) !=head);

set\_page\_writeback(page);

do {

struct buffer\_head \*next = bh->b\_this\_page;

if(buffer\_async\_write(bh)) {

submit\_bh\_wbc(REQ\_OP\_WRITE,write\_flags,bh,inode->i\_write\_hint,wbc);

bh = next;

}

}while(bh!=head)

}

create\_page\_buffers(page,inode,(1<<BH\_Dirty) | (1 << BH\_Uptodate))

{

/\*

page\_has\_buffers通过struct page的private成员判断是否为空来判断

\*/

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

create\_empty\_buffers(page,1<<READ\_ONCE(inode->i\_blkbits), (1<<BH\_Dirty) | (1 << BH\_Uptodate));

return page\_buffers(page);

}

create\_empty\_buffers(page,1<<READ\_ONCE(inode->i\_blkbits), (1<<BH\_Dirty) | (1 << BH\_Uptodate))

{

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

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

bh = head;

/\*

这段代码的作用是将struct buffer\_head串联成环形缓冲区

\*/

do {

bh->b\_state |= (1<<BH\_Dirty) | (1 <<BH\_Uptodate);

tail = bh;

bh = bh->b\_this\_page;

}while(bh);

tail->b\_this\_page = head;

attach\_page\_buffers(page,head);

}

alloc\_page\_buffers(page,blocksize,true)

{

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

gfp\_t gfp = GFP\_NOFS | \_\_GFP\_ACCOUNT;

offset = PAGE\_SIZE;

head = NULL;

/\*

struct buffer\_head 通过b\_this\_page成员将struct buffer\_head串联起来

bh0->bh1-> bh2->bh3(offset =3 \*block\_size)->NULL

\*/

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

/\*

alloc\_buffer\_head ，通过kmem\_cache\_zalloc和bh\_cachep分配struct buffer\_head结构体

\*/

bh = alloc\_buffer\_head(gfp);

if(!bh)

goto no\_grow;

bh->b\_this\_page =head;

bh->b\_blocknr = -1;

head = bh;

bh->b\_size = size;

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

}

}

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

{

bh->b\_page =page;

if(PageHighMem(page))

bh->b\_data = (char \*)(0+offset);

else

bh->b\_data = page\_address(page) +offset;

}

attach\_page\_buffers(page,head) {

/\*

增加page的引用计数

\*/

get\_page(page);

SetPagePrivate(page);

set\_page\_private(page,(unsigned long )head);

}

submit\_bh\_wbc(REQ\_OP\_WRITE,write\_flags,bh,inode->i\_write\_hint,wbc)

{

struct bio \*bio;

if(test\_set\_buffer\_req(bh)&&(op == REQ\_OP\_WRITE))

clear\_buffer\_write\_io\_error(bh);

/\*

from here on down,it is all bio –do the initial mapping,

submit\_bio -> generic\_make\_request may further map this bio around

\*/

bio = bio\_alloc(GFP\_NOIO,1);

}

clean\_buffers(page,first\_unmapped)

{

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

unsigned buffer\_count =0;

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

return;

head = page\_buffers(page);

bh = head;

do {

if(buffer\_counter++ == first\_unmapped)

break;

clear\_buffer\_dirty(bh);

bh = bh->b\_this\_page;

}while(bh !=head);

if(buffer\_heads\_over\_limit &&PageUptodate(page))

try\_to\_free\_buffers(page);

}