

# Linux Disk Caches

Wu Chia-Chih  
Linux Kernel Trace  
Seminar  
2002.6.17



# Outline

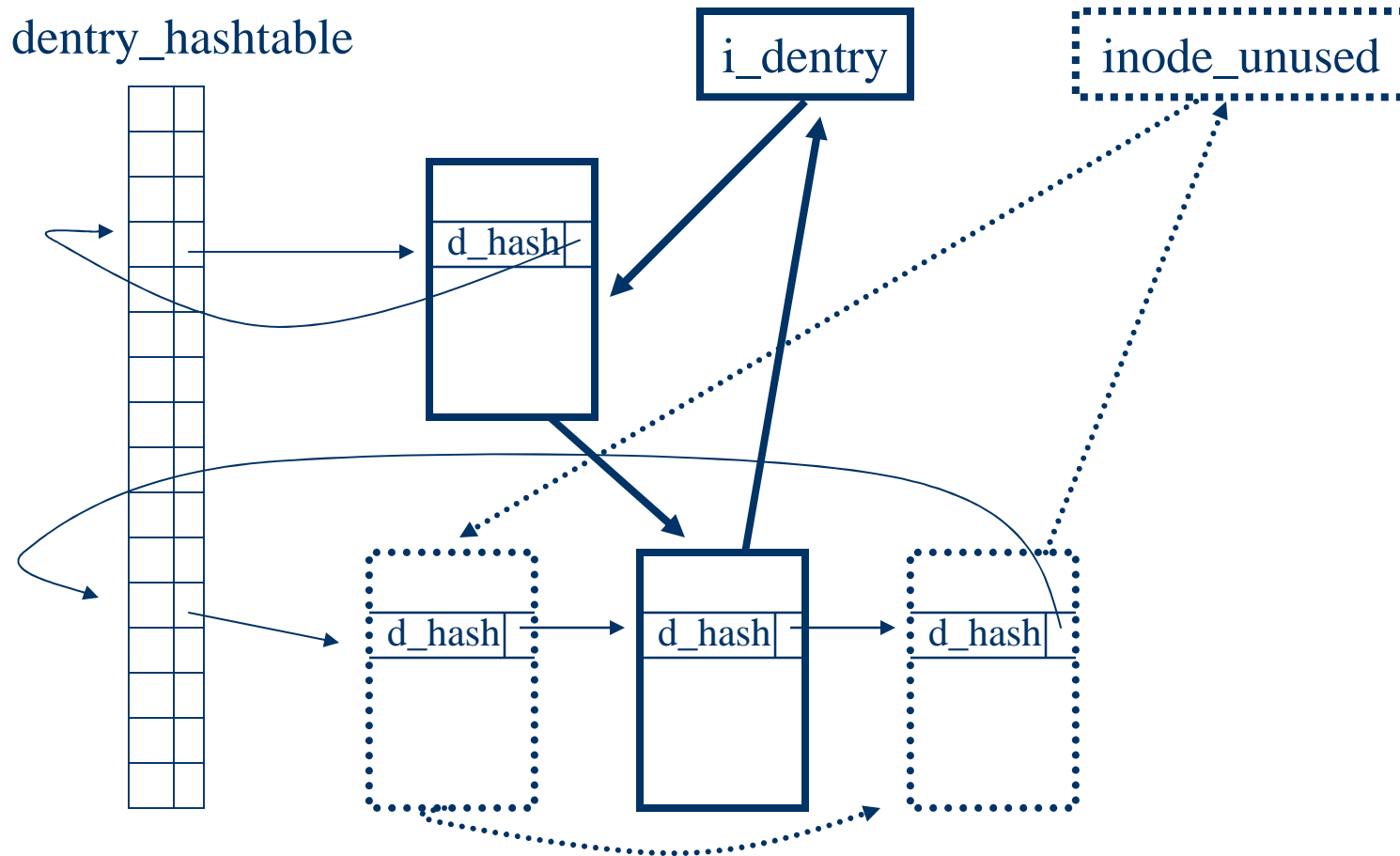
---

- I Introduction
- I Dentry cache
- I Buffer cache
  - Usage of buffer cache
  - Flush the buffer cache
- I Page cache

# Introduction

- | a software mechanism that allows the system to keep in RAM some data normally stored on a disk
- | further accesses to that data can be satisfied quickly without accessing the disk
- | Dentry cache : speed up the translation of a file pathname to the corresponding inode
- | Buffer cache : a disk cache that stores buffers
- | Page cache : a disk cache storing page frames that contain data belonging to regular files

# Dentry cache



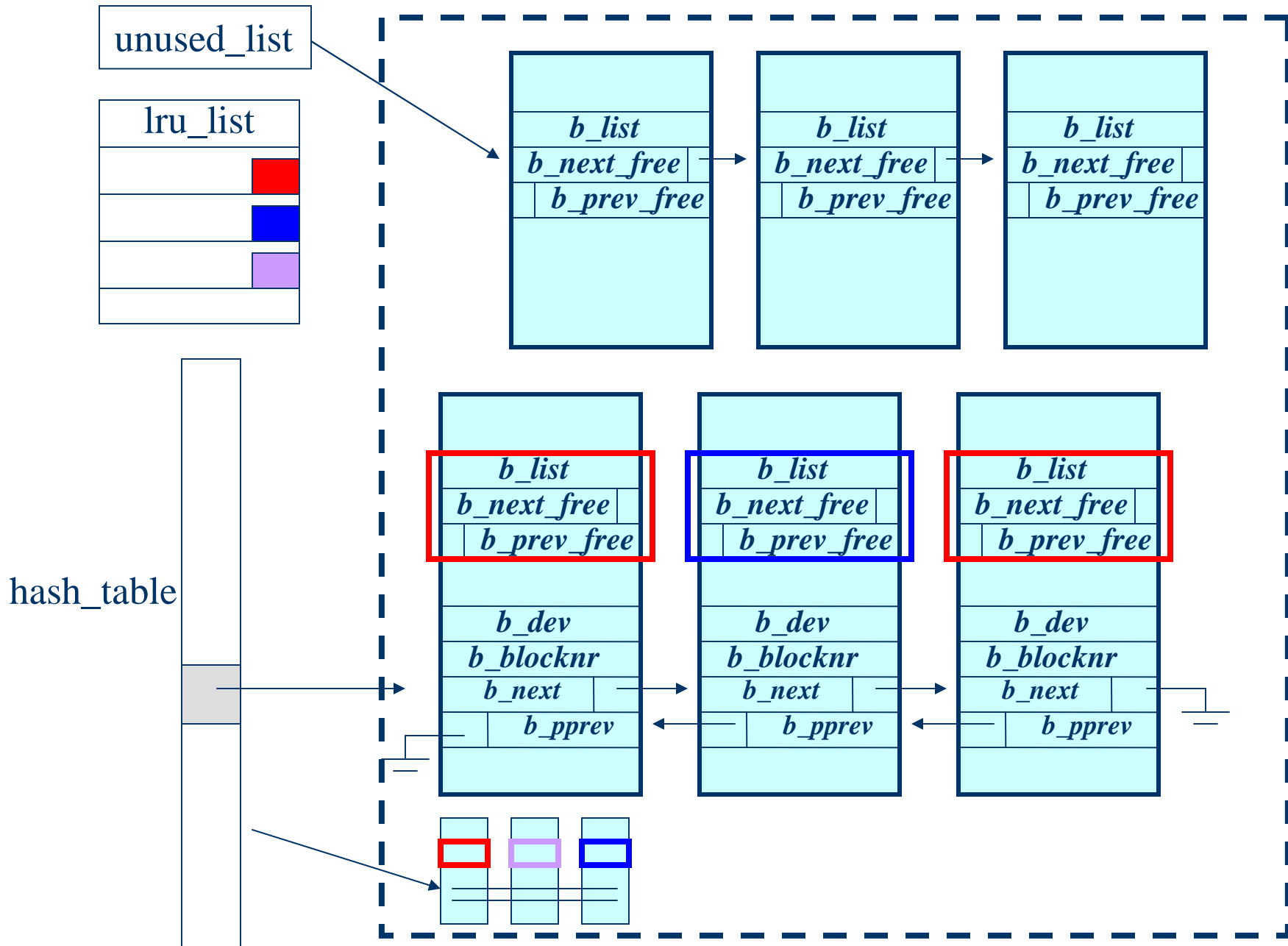
# Buffer cache

## I Data structure

- A set of buffer heads (***struct buffer\_head\****) describing the buffers
- A hash table help the kernel quickly derive the buffer head that describes the buffer associated with a given pair of device and block numbers

## Global Variables

## Available Buffer Caches



# The `b_state` field

- | `BH_Uptodate` */\* the buffer contains valid data \*/*
- | `BH_Dirty` */\* the buffer contains data that must be written to the block device \*/*
- | `BH_Lock` */\* the buffer is involved in a disk transfer \*/*
- | `BH_Req` */\* the corresponding block has been requested and has valid (up-to-date) data \*/*
- | `BH_Protected` */\* the buffer is protected (protected buffers never get freed). Used only to implement RAM disks on top of the buffer cache \*/*

# The b\_list field(three different lists)

## I BUF\_CLEAN list

- Collects nondirty buffers (BH\_Dirty is off)
- Not necessarily to be up-to-date

## I BUF\_DIRTY list

- Collects buffer heads of dirty buffers that have not been written to disk (BH\_Dirty is on;BH\_Lock is off)

## I BUF\_LOCKED list

- Collects buffers heads of dirty buffers that have been selected to be written to disk (BH\_Lock is on;BH\_Dirty is clear)

# struct buffer\_head

```
struct buffer_head {  
    /* First cache line: */  
    struct buffer_head *b_next;    /* Hash queue list */  
    unsigned long b_blocknr;    /* block number */  
    unsigned short b_size;    /* block size */  
    unsigned short b_list;    /* List that this buffer appears */  
    kdev_t b_dev;    /* device (B_FREE = free) */  
  
    atomic_t b_count;    /* users using this block */  
    kdev_t b_rdev;    /* Real device */  
    unsigned long b_state;    /* buffer state bitmap (see above) */  
    unsigned long b_flushtime;    /* Time when (dirty) buffer should be written */  
};
```

```

struct buffer_head *b_next_free; /* lru/free list linkage */
struct buffer_head *b_prev_free; /* doubly linked list of buffers */
struct buffer_head *b_this_page; /* circular list of buffers in one page */
struct buffer_head *b_reqnext; /* request queue */

struct buffer_head **b_pprev; /* doubly linked list of hash-queue */
char * b_data; /* pointer to data block */
struct page *b_page; /* the page this bh is mapped to */
void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O completion */
void *b_private; /* reserved for b_end_io */

unsigned long b_rsector; /* Real buffer location on disk */
wait_queue_head_t b_wait;

struct inode * b_inode;
struct list_head b_inode_buffers; /* doubly linked list of inode dirty buffers */

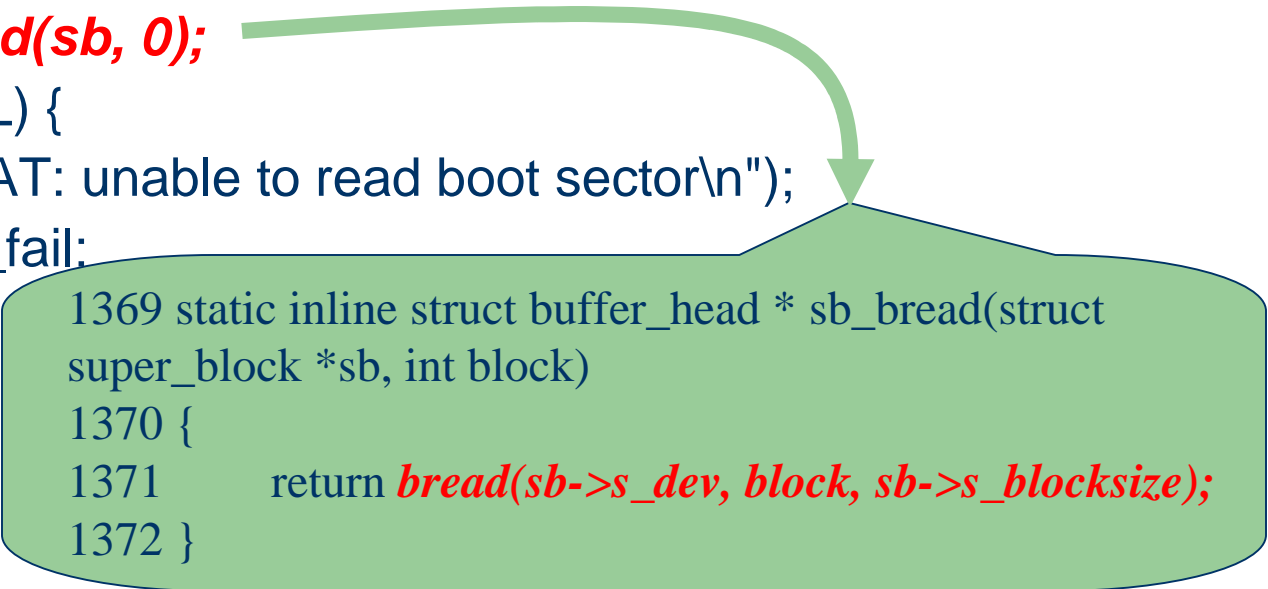
};

```

# Usage of Buffer Cache

```
544 struct super_block *
545 fat_read_super(struct super_block *sb, void *data, int silent,
546               struct inode_operations *fs_dir_inode_ops)
547 {
    .....
587     bh = sb_bread(sb, 0);
588     if (bh == NULL) {
589         printk("FAT: unable to read boot sector\n");
590         goto out_fail;
591     }

```



```
1369 static inline struct buffer_head * sb_bread(struct
super_block *sb, int block)
1370 {
1371     return bread(sb->s_dev, block, sb->s_blocksize);
1372 }
```

# bread(kdev\_t dev, int block, int size)

- Reads a specified block, and returns buffer head that contains it.

```
1185     bh = getblk(dev, block, size);
1186     touch_buffer(bh);
1187     if (buffer_uptodate(bh))
1188         return bh;
1189     ll_rw_block(READ, 1, &bh);
1190     wait_on_buffer(bh);
1191     if (buffer_uptodate(bh))
1192         return bh;
1193     brelse(bh);
1194     return NULL;
```



Buffer  
Allocation

Main service routine for the buffer cache

## Buffer Allocation - the getblk() Function

```
1027 struct buffer_head * getblk(kdev_t dev, int block, int size)
1028 {
1029     for (;;) {
1030         struct buffer_head * bh;
1031         bh = get_hash_table(dev, block, size);
1032         if (bh)
1033             return bh;
1034
1035         if (!grow_buffers(dev, block, size))
1036             free_more_memory();
1037     }
1038 }
1039 }
```

*Already  
in cache*

*get a proper block  
from cache*

*Try to get  
more space  
from cache*

```
hash_table[(_hashfn(HASHDEV(dev),block) & bh_hash_mask)]
```

## get\_hash\_table(dev,block,size)

```
563     struct buffer_head *bh, **p = &hash(dev, block);
564
565     read_lock(&hash_table_lock);
566
567     for (;;) {
568         bh = *p;
569         if (!bh)
570             break;
571         p = &bh->b_next;
572         if (bh->b_blocknr != block)
573             continue;
```

*Check block  
number*

# get\_hash\_table(dev,block,size)

```
574     if (bh->b_size != size)
575         continue;
576     if (bh->b_dev != dev)
577         continue;
578     get_bh(bh);
579     break;
580 }
581
582 read_unlock(&hash_table_lock);
583 return bh;
584 }
```

*Check size  
and device  
identifier*

# grow\_buffers(dev,block,size)

```
2552 struct page * page;  
2553 struct block_device *bdev;  
2554 unsigned long index;  
2555 int sizebits;
```

```
2556  
2557  
2558  
2559  
2560  
2561  
2562  
2563
```

```
/* Size must be multiple of hard sectorsize */  
if (size & (get_hardsect_size(dev)-1))  
    BUG();  
/* Size must be within 512 bytes and PAGE_SIZE */  
if (size < 512 || size > PAGE_SIZE)  
    BUG();
```

Check size

# grow\_buffers(dev,block,size)

```
2564     sizebits = -1;
2565     do {
2566         sizebits++;
2567     } while ((size << sizebits) < PAGE_SIZE);
2569     index = block >> sizebits;
2570     block = index << sizebits;
2571
2572     bdev = bdget(kdev_t_to_nr(dev));
2573     if (!bdev) {
2574         printk("No block device for %s\n", kdevname(dev));
2575         BUG();
2576     }
2577
2578     /* Create a page with the proper size buffers.. */
2579     page = grow_dev_page(bdev, index, size);
```

# grow\_buffers(dev,block,size)

```
2581     /* This is "wrong" - talk to Al Viro */
2582     atomic_dec(&bdev->bd_count);
2583     if (!page)
2584         return 0;
2585
2586     /* Hash in the buffers on the hash list */
2587     hash_page_buffers(page, dev, block, size);
2588     UnlockPage(page);
2589     page_cache_release(page);
2590
2591     /* We hashed up this page, so increment buffermem */
2592     atomic_inc(&buffermem_pages);
2593     return 1;
2594 }
```

# grow\_dev\_page()

*Find page in  
page caches or  
allocate a new  
page*

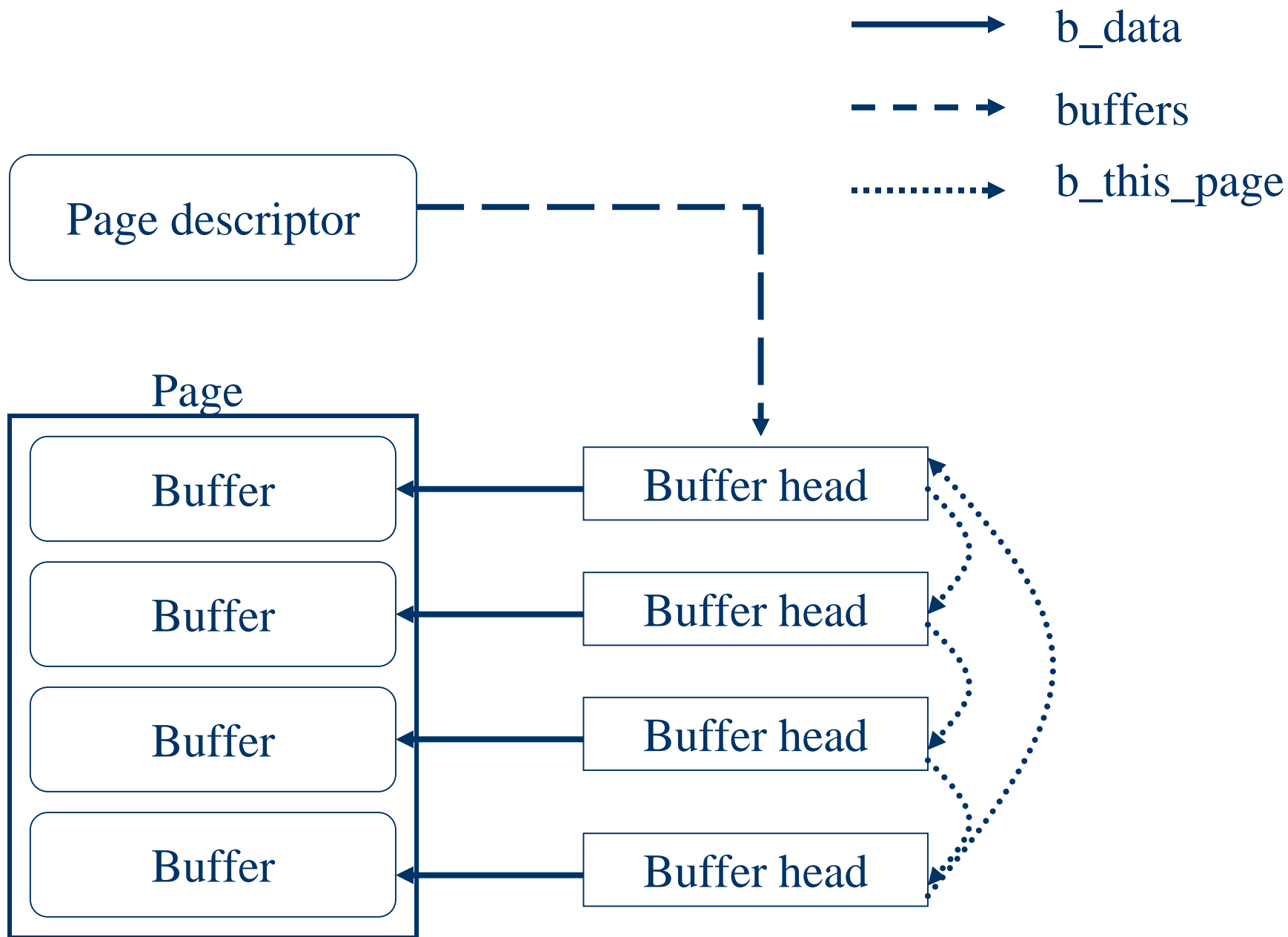
```
2486 {
2487     struct page * page;
2488     struct buffer_head *bh;
2489
2490     page = find_or_create_page(bdev->bd_inode->i_mapping, index, GFP_NOFS);
2491     if (!page)
2492         return NULL;
2493
2494     if (!PageLocked(page))
2495         BUG();
2496
2497     bh = page->buffers;
2498     if (bh) {
2499         if (bh->b_size == size)
2500             return page;
```

# grow\_dev\_page()

```
2501         if (!try_to_free_buffers(page, GFP_NOFS))
2502             goto failed;
2503     }
2504
2505     bh = create_buffers(page, size, 0);
2506     if (!bh)
2507         goto failed;
2508     link_dev_buffers(page, bh);
2509     return page;
2510
2511 failed:
2512     UnlockPage(page);
2513     page_cache_release(page);
2514     return NULL;
2515 }
```

*Creates (PAGE\_SIZE / “size”) buffers in “page”*

*Link the created bh’s into a circular linked-list by “b\_this\_page” element*



# create\_buffers()

```
1298 {  
    .....  
1302 try_again:  
1303     head = NULL;  
1304     offset = PAGE_SIZE;  
1305     while ((offset -= size) >= 0) {  
1306         bh = get_unused_buffer_head(async);  
        .....  
1311         bh->b_this_page = head;  
1312         head = bh;  
        .....  
1320         set_bh_page(bh, page, offset);  
        .....  
1324     }  
1325     return head;
```

*Connect the bhs  
in a linked-list*

Case 1: # of unused buffer > # of reserved buffer

## get\_unused\_buffer\_head()

```
struct buffer_head * bh;  
spin_lock(&unused_list_lock);  
if (nr_unused_buffer_heads > NR_RESERVED) {  
    bh = unused_list;  
    unused_list = bh->b_next_free;  
    nr_unused_buffer_heads--;  
    spin_unlock(&unused_list_lock);  
    return bh;  
}  
spin_unlock(&unused_list_lock);
```

- NR\_RESERVED (usually 16) elements in the list is reserved for page I/O operations.
- This is done to prevent nasty deadlocks caused by the lack of free buffer heads.

Case 2: allocate a new buffer head

## get\_unused\_buffer\_head()

.....

```
if((bh = kmem_cache_alloc(bh_cache, SLAB_NOFS)) != NULL) {  
    bh->b_blocknr = -1;  
    bh->b_this_page = NULL;  
    return bh;  
}
```

.....

Case 3: if we need a async IO buffer, use the reserved buffer

## get\_unused\_buffer\_head()

.....

```
if (async) {  
    spin_lock(&unused_list_lock);  
    if (unused_list) {  
        bh = unused_list;  
        unused_list = bh->b_next_free;  
        nr_unused_buffer_heads--;  
        spin_unlock(&unused_list_lock);  
        return bh;  
    }  
}
```

# bdflush()

```
2950     for (;;) {
2951         CHECK_EMERGENCY_SYNC
2952
2953         spin_lock(&lru_list_lock);
2954         if (!write_some_buffers(NODEV) || balance_dirty_state() < 0) {
2955             wait_for_some_buffers(NODEV);
2956             interruptible_sleep_on(&bdflush_wait);
2957         }
2958     }
```

# balance\_dirty()

```
1073 void balance_dirty(void)
1074 {
1075     int state = balance_dirty_state();
1076
1077     if (state < 0)
1078         return;
1079
1080     .....
1082     write_some_buffers(NODEV);
1083
1084     .....
1090     if (state > 0) {
1091         wait_for_some_buffers(NODEV);
1092         wakeup_bdflush();
1093     }
```

*-1 not dirty*  
*0 dirty but OK*  
*1 really dirty*

# write\_some\_buffers()

```
191 static int write_some_buffers(kdev_t dev)
192 {
```

```
.....
```

```
198     next = lru_list[BUF_DIRTY];
199     nr = nr_buffers_type[BUF_DIRTY];
200     count = 0;
201     while (next && --nr >= 0) {
202         struct buffer_head * bh = next;
203         next = bh->b_next_free;
```

```
204
205         if (dev && bh->b_dev != dev)
206             continue;
207         if (test_and_set_bit(BH_Lock, &bh->b_state))
208             continue;
```

Guarantee that  
this “bh” is in the  
right dev and not  
locked

# write\_some\_buffers()

```
209     if (atomic_set_buffer_clean(bh)) {
210         __refile_buffer(bh);
211         get_bh(bh);
212         array[count++] = bh;
213         if (count < NRSYNC)
214             continue;
215
216         spin_unlock(&lru_list_lock);
217         write_locked_buffers(array, count);
218         return -EAGAIN;
219     }
220     unlock_buffer(bh);
221     __refile_buffer(bh);
222 }
```

if( bh is dirty )

Collects utmost  
NRSYNC "bhs"  
in array[]

Do the flush job

## write\_locked\_buffers()

```
176 {
177     do {
178         struct buffer_head * bh = *array++;
179         bh->b_end_io = end_buffer_io_sync;
180         submit_bh(WRITE, bh);
181     } while (--count);
182 }
```

# Page Cache

## I the page hash table

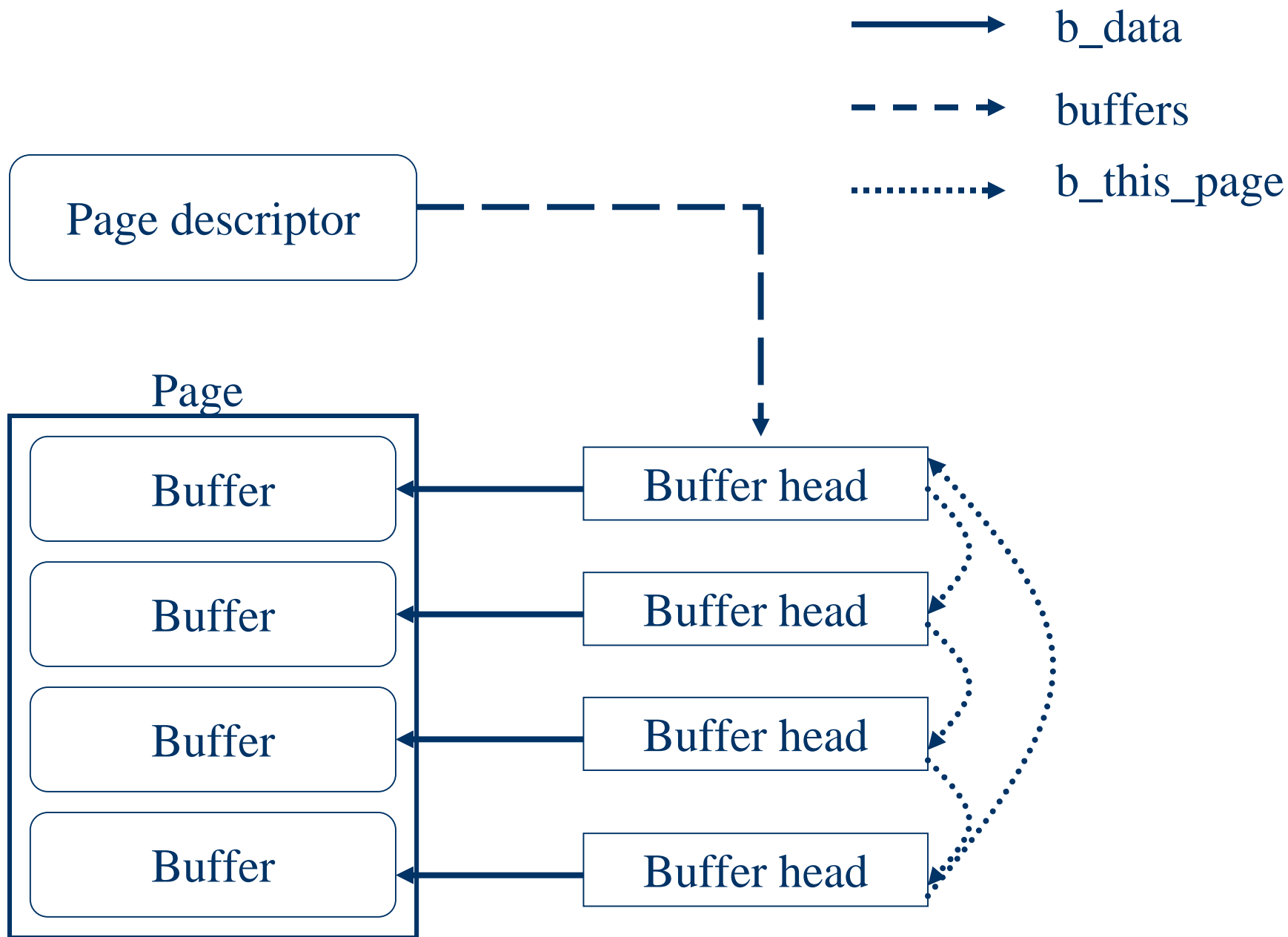
- Lets the kernel quickly derive the page descriptor address for the page
- `page_hash(inode->i_mapping, file offset)`
- `page = __find_page_nolock(mapping, index, *hash);`

## I the inode queue

- A list of page descriptors corresponding to pages of data of a particular file

# Page descriptor fields related to the page cache

```
151 typedef struct page {  
153     struct address_space *mapping;  
154     unsigned long index;  
155     struct page *next_hash;  
     .....  
160     struct list_head lru;  
     .....  
162     wait_queue_head_t wait;  
163     struct page **pprev_hash;  
164     struct buffer_head * buffers;  
     .....  
168 } mem_map_t;
```



# do\_generic\_file\_read()

```
1253 void do_generic_file_read(struct file * filp, loff_t *ppos, read_descriptor_t * desc,
1254   read_actor_t actor)
1255     struct address_space *mapping = filp->f_dentry->d_inode->i_mapping;
.....
1323     /*
1324     * Try to find the data in the page cache..
1325     */
1326     hash = page_hash(mapping, index);
1327
1328     spin_lock(&pagecache_lock);
1329     page = __find_page_nolock(mapping, index, *hash);
1330     if (!page)
1331         goto no_cached_page;
```

# do\_generic\_file\_read() //page not found

1420 no\_cached\_page:

```
.....
1427         if (!cached_page) {
1428             spin_unlock(&pagecache_lock);
1429             cached_page = page_cache_alloc(mapping);
1430             if (!cached_page) {
1431                 desc->error = -ENOMEM;
1432                 break;
1433             }
.....
1439             spin_lock(&pagecache_lock);
1440             page = __find_page_nolock(mapping, index, *hash);
1441             if (page)
1442                 goto found_page;
1443         }
```

## do\_generic\_file\_read() //page not found

```
1444
1445     /*
1446     * Ok, add the new page to the hash-queues...
1447     */
1448     page = cached_page;
1449     __add_to_page_cache(page, mapping, index, hash);
1450     spin_unlock(&pagecache_lock);
1451     lru_cache_add(page);
1452     cached_page = NULL;
1453
1454     goto readpage;
```

## \_\_add\_to\_page\_cache()

```
651 {
652     unsigned long flags;
653
654     flags = page->flags & ~(1 << PG_uptodate | 1 << PG_error | 1 <<
PG_dirty | 1 << PG_referenced | 1 << PG_arch_1 | 1 << PG_checked);
655     page->flags = flags | (1 << PG_locked);
656     page_cache_get(page);
657     page->index = offset;
658     add_page_to_inode_queue(mapping, page);
659     add_page_to_hash_queue(page, hash);
660 }
```

# add\_page\_to\_inode\_queue()

```
86 static inline void add_page_to_inode_queue(struct address_space
    *mapping, struct page * page)
87 {
88     struct list_head *head = &mapping->clean_pages;
89
90     mapping->nropages++;
91     list_add(&page->list, head);
92     page->mapping = mapping;
93 }
```

# add\_page\_to\_hash\_queue()

```
71 static void FASTCALL(add_page_to_hash_queue(struct page * page, struct page
    **p));
72 static void add_page_to_hash_queue(struct page * page, struct page **p)
73 {
74     struct page *next = *p;
75
76     *p = page;
77     page->next_hash = next;
78     page->pprev_hash = p;
79     if (next)
80         next->pprev_hash = &page->next_hash;
81     if (page->buffers)
82         PAGE_BUG(page);
83     atomic_inc(&page_cache_size);
84 }
```

# struct address\_space\_operations

```
379 struct address_space_operations {
380     int (*writepage)(struct page *);
381     int (*readpage)(struct file *, struct page *);
382     int (*sync_page)(struct page *);
387     int (*prepare_write)(struct file *, struct page *, unsigned, unsigned);
388     int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
390     int (*bmap)(struct address_space *, long);
395 };
```

## do\_generic\_file\_read()

```
1399 readpage:
1400     /* ... and start the actual read. The read will unlock the page. */
1401     error = mapping->a_ops->readpage(filp, page);
1402
1403     if (!error) {
1404         if (Page_Uptodate(page))
1405             goto page_ok;
```

# Reference

- | Understanding the LINUX KERNEL - *O'reilly*
- | Linux Kernel 2.4 Internals
  - *<http://www.tldp.org/LDP/lki/lki.html>*
- | Cross-Reference Linux - *<http://lxr.linux.no>*

```
263 static struct dentry * cached_lookup(struct dentry * parent, struct qstr * name, int flags)
264 {
265     struct dentry * dentry = d_lookup(parent, name);
266
267     if (dentry && dentry->d_op && dentry->d_op->d_revalidate) {
268         if (!dentry->d_op->d_revalidate(dentry, flags) && !d_invalidate(dentry)) {
269             dput(dentry);
270             dentry = NULL;
271         }
272     }
273     return dentry;
274 }
```