```
 1 /*
 2  *  linux/fs/buffer.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  *  'buffer.c' implements the buffer-cache functions. Race-conditions have
 9  * been avoided by NEVER letting a interrupt change a buffer (except for the
10  * data, of course), but instead letting the caller do it. NOTE! As interrupts
11  * can wake up a caller, some cli-sti sequences are needed to check for
12  * sleep-on-calls. These should be extremely quick, though (I hope).
13  */
/*
 * 'buffer.c'用于实现缓冲区高速缓存功能。通过不让中断处理过程改变缓冲区，而是让调
 * 用者来执行，避免了竞争条件（当然除改变数据以外）。注意！由于中断可以唤醒一个调
 * 用者，因此就需要开关中断指令（cli-sti）序列来检测由于调用而睡眠。但需要非常地快
 * （我希望是这样）。
 */
14
15 /*
16  * NOTE! There is one discordant note here: checking floppies for
17  * disk change. This is where it fits best, I think, as it should
18  * invalidate changed floppy-disk-caches.
19  */
/*
 * 注意！有一个程序应不属于这里：检测软盘是否更换。但我想这里是放置
 * 该程序最好的地方了，因为它需要使已更换软盘缓冲失效。
 */
20
21 #include <stdarg.h>       // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                             // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                             // vsprintf、vprintf、vfprintf 函数。
22
23 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
24 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
25 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
26 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入汇编宏。
27 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
28
// 变量 end 是由编译时的连接程序 ld 生成，用于表明内核代码的末端，即指明内核模块末端
// 位置，参见错误!未找到引用源。。也可以从编译内核时生成的 System.map 文件中查出。这里用它来表
// 明高速缓冲区开始于内核代码末端位置。
// 第 33 行上的 buffer_wait 变量是等待空闲缓冲块而睡眠的任务队列头指针。它与缓冲块头
// 部结构中 b_wait 指针的作用不同。当任务申请一个缓冲块而正好遇到系统缺乏可用空闲缓
// 冲块时，当前任务就会被添加到 buffer_wait 睡眠等待队列中。而 b_wait 则是专门供等待
// 指定缓冲块（即 b_wait 对应的缓冲块）的任务使用的等待队列头指针。
29 extern int end;
30 struct buffer_head * start_buffer = (struct buffer_head *) &end;
31 struct buffer_head * hash_table[NR_HASH];        // NR_HASH = 307 项。
32 static struct buffer_head * free_list;           // 空闲缓冲块链表头指针。
```

```
33 static struct task_struct * buffer_wait = NULL;  // 等待空闲缓冲块而睡眠的任务队列。

   // 下面定义系统缓冲区中含有的缓冲块个数。这里，NR_BUFFERS 是一个定义在 linux/fs.h 头
   // 文件第 48 行的宏，其值即是变量名 nr_buffers，并且在 fs.h 文件第 172 行声明为全局变量。
   // 大写名称通常都是一个宏名称，Linus 这样编写代码是为了利用这个大写名称来隐含地表示
   // nr_buffers 是一个在内核初始化之后不再改变的“常量”。它将在初始化函数 buffer_init()
   // 中被设置（第 371 行）。
34 int NR_BUFFERS = 0;                                  // 系统含有缓冲块个数。
35
   //// 等待指定缓冲块解锁。
   // 如果指定的缓冲块 bh 已经上锁就让进程不可中断地睡眠在该缓冲块的等待队列 b_wait 中。
   // 在缓冲块解锁时，其等待队列上的所有进程将被唤醒。虽然是在关闭中断（cli）之后去睡
   // 眠的，但这样做并不会影响在其他进程上下文中响应中断。因为每个进程都在自己的 TSS 段
   // 中保存了标志寄存器 EFLAGS 的值，所以在进程切换时 CPU 中当前 EFLAGS 的值也随之改变。
   // 使用 sleep_on()进入睡眠状态的进程需要用 wake_up()明确地唤醒。
36 static inline void wait_on_buffer(struct buffer_head * bh)
37 {
38         cli();                                       // 关中断。
39         while (bh->b_lock)                           // 如果已被上锁则进程进入睡眠，等待其解锁。
40                 sleep_on(&bh->b_wait);
41         sti();                                       // 开中断。
42 }
43
   //// 设备数据同步。
   // 同步设备和内存高速缓冲中数据。其中，sync_inodes()定义在 inode.c，59 行。
44 int sys_sync(void)
45 {
46         int i;
47         struct buffer_head * bh;
48
   // 首先调用 i 节点同步函数，把内存 i 节点表中所有修改过的 i 节点写入高速缓冲中。然后
   // 扫描所有高速缓冲区，对已被修改的缓冲块产生写盘请求，将缓冲中数据写入盘中，做到
   // 高速缓冲中的数据与设备中的同步。
49         sync_inodes();                               /* write out inodes into buffers */
50         bh = start_buffer;                           // bh 指向缓冲区开始处。
51         for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
52                 wait_on_buffer(bh);                  // 等待缓冲区解锁（如果已上锁的话）。
53                 if (bh->b_dirt)
54                         ll_rw_block(WRITE,bh);       // 产生写设备块请求。
55         }
56         return 0;
57 }
58
   //// 对指定设备进行高速缓冲数据与设备上数据的同步操作。
   // 该函数首先搜索高速缓冲区中所有缓冲块。对于指定设备 dev 的缓冲块，若其数据已被修改
   // 过就写入盘中（同步操作）。然后把内存中 i 节点表数据写入高速缓冲中。之后再对指定设
   // 备 dev 执行一次与上述相同的写盘操作。
59 int sync_dev(int dev)
60 {
61         int i;
62         struct buffer_head * bh;
63
   // 首先对参数指定的设备执行数据同步操作，让设备上的数据与高速缓冲区中的数据同步。
```

```
    // 方法是扫描高速缓冲区中所有缓冲块，对指定设备 dev 的缓冲块，先检测其是否已被上锁，
    // 若已被上锁就睡眠等待其解锁。然后再判断一次该缓冲块是否还是指定设备的缓冲块并且
    // 已修改过（b_dirt 标志置位），若是就对其执行写盘操作。因为在我们睡眠期间该缓冲块
    // 有可能已被释放或者被挪作它用，所以在继续执行前需要再次判断一下该缓冲块是否还是
    // 指定设备的缓冲块，
64          bh = start_buffer;                      // bh 指向缓冲区开始处。
65          for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
66                  if (bh->b_dev != dev)           // 不是设备 dev 的缓冲块则继续。
67                          continue;
68                  wait_on_buffer(bh);             // 等待缓冲区解锁（如果已上锁的话）。
69                  if (bh->b_dev == dev && bh->b_dirt)
70                          ll_rw_block(WRITE,bh);
71          }
    // 再将 i 节点数据写入高速缓冲。让 i 节点表 inode_table 中的 inode 与缓冲中的信息同步。
72          sync_inodes();
    // 然后在高速缓冲中的数据更新之后，再把它们与设备中的数据同步。这里采用两遍同步操作
    // 是为了提高内核执行效率。第一遍缓冲区同步操作可以让内核中许多“脏块”变干净，使得
    // i 节点的同步操作能够高效执行。本次缓冲区同步操作则把那些由于 i 节点同步操作而又变
    // 脏的缓冲块与设备中数据同步。
73          bh = start_buffer;
74          for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
75                  if (bh->b_dev != dev)
76                          continue;
77                  wait_on_buffer(bh);
78                  if (bh->b_dev == dev && bh->b_dirt)
79                          ll_rw_block(WRITE,bh);
80          }
81          return 0;
82  }
83
    //// 使指定设备在高速缓冲区中的数据无效。
    // 扫描高速缓冲区中所有缓冲块。对指定设备的缓冲块复位其有效(更新)标志和已修改标志。
84  void inline invalidate_buffers(int dev)
85  {
86          int i;
87          struct buffer_head * bh;
88
89          bh = start_buffer;
90          for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
91                  if (bh->b_dev != dev)           // 如果不是指定设备的缓冲块，则
92                          continue;               // 继续扫描下一块。
93                  wait_on_buffer(bh);             // 等待该缓冲区解锁（如果已被上锁）。
    // 由于进程执行过睡眠等待，所以需要再判断一下缓冲区是否是指定设备的。
94                  if (bh->b_dev == dev)
95                          bh->b_uptodate = bh->b_dirt = 0;
96          }
97  }
98
99  /*
100  * This routine checks whether a floppy has been changed, and
101  * invalidates all buffer-cache-entries in that case. This
102  * is a relatively slow routine, so we have to try to minimize using
103  * it. Thus it is called only upon a 'mount' or 'open'. This
```

```
104  * is the best way of combining speed and utility, I think.
105  * People changing diskettes in the middle of an operation deserve
106  * to loose :-)
107  *
108  * NOTE! Although currently this is only for floppies, the idea is
109  * that any additional removable block-device will use this routine,
110  * and that mount/open needn't know that floppies/whatever are
111  * special.
112  */
     /*
      * 该子程序检查一个软盘是否已被更换，如果已经更换就使高速缓冲中与该软驱
      * 对应的所有缓冲区无效。该子程序相对来说较慢，所以我们要尽量少使用它。
      * 所以仅在执行'mount'或'open'时才调用它。我想这是将速度和实用性相结合的
      * 最好方法。若在操作过程中更换软盘，就会导致数据的丢失。这是咎由自取☺。
      *
      * 注意！尽管目前该子程序仅用于软盘，以后任何可移动介质的块设备都将使用该
      * 程序，mount/open 操作不需要知道是软盘还是其他什么特殊介质。
      */
     //// 检查磁盘是否更换，如果已更换就使对应高速缓冲区无效。
113  void check_disk_change(int dev)
114  {
115          int i;
116
     // 首先检测一下是不是软盘设备。因为现在仅支持软盘可移动介质。如果不是则退出。然后
     // 测试软盘是否已更换，如果没有则退出。floppy_change()在 blk_drv/floppy.c 第 139 行。
117          if (MAJOR(dev) != 2)
118                  return;
119          if (!floppy_change(dev & 0x03))
120                  return;
     // 软盘已经更换，所以释放对应设备的 i 节点位图和逻辑块位图所占的高速缓冲区；并使该
     // 设备的 i 节点和数据块信息所占踞的高速缓冲块无效。
121          for (i=0 ; i<NR_SUPER ; i++)
122                  if (super_block[i].s_dev == dev)
123                          put_super(super_block[i].s_dev);
124          invalidate_inodes(dev);
125          invalidate_buffers(dev);
126  }
127
     // 下面两行代码是 hash（散列）函数定义和 hash 表项的计算宏。
     // hash 表的主要作用是减少查找比较元素所花费的时间。通过在元素的存储位置与关键字之间
     // 建立一个对应关系（hash 函数），我们就可以直接通过函数计算立刻查询到指定的元素。建
     // 立 hash 函数的指导条件主要是尽量确保散列到任何数组项的概率基本相等。建立函数的方法
     // 有多种，这里 Linux 0.12 主要采用了关键字除留余数法。因为我们寻找的缓冲块有两个条件，
     // 即设备号 dev 和缓冲块号 block，因此设计的 hash 函数肯定需要包含这两个关键值。这里两个
     // 关键字的异或操作只是计算关键值的一种方法。再对关键值进行 MOD 运算就可以保证函数所计
     // 算得到的值都处于函数数组项范围内。
128  #define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
129  #define hash(dev,block) hash_table[_hashfn(dev,block)]
130
     //// 从 hash 队列和空闲缓冲队列中移走缓冲块。
     // hash 队列是双向链表结构，空闲缓冲块队列是双向循环链表结构。
131  static inline void remove_from_queues(struct buffer_head * bh)
132  {
```

```
133 /* remove from hash-queue */
    /* 从 hash 队列中移除缓冲块 */
134         if (bh->b_next)
135                 bh->b_next->b_prev = bh->b_prev;
136         if (bh->b_prev)
137                 bh->b_prev->b_next = bh->b_next;
    // 如果该缓冲区是该队列的头一个块，则让 hash 表的对应项指向本队列中的下一个缓冲区。
138         if (hash(bh->b_dev,bh->b_blocknr) == bh)
139                 hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
140 /* remove from free list */
    /* 从空闲缓冲块表中移除缓冲块 */
141         if (!(bh->b_prev_free) || !(bh->b_next_free))
142                 panic("Free block list corrupted");
143         bh->b_prev_free->b_next_free = bh->b_next_free;
144         bh->b_next_free->b_prev_free = bh->b_prev_free;
    // 如果空闲链表头指向本缓冲区，则让其指向下一缓冲区。
145         if (free_list == bh)
146                 free_list = bh->b_next_free;
147 }
148
    //// 将缓冲块插入空闲链表尾部，同时放入 hash 队列中。
149 static inline void insert_into_queues(struct buffer_head * bh)
150 {
151 /* put at end of free list */
    /* 放在空闲链表末尾处 */
152         bh->b_next_free = free_list;
153         bh->b_prev_free = free_list->b_prev_free;
154         free_list->b_prev_free->b_next_free = bh;
155         free_list->b_prev_free = bh;
156 /* put the buffer in new hash-queue if it has a device */
    /* 如果该缓冲块对应一个设备，则将其插入新 hash 队列中 */
    // 请注意当 hash 表某项第 1 次插入项时，hash()计算值肯定为 NULL，因此此时第 161 行上
    // 得到的 bh->b_next 肯定是 NULL，所以第 163 行上应该在 bh->b_next 不为 NULL 时才能给
    // b_prev 赋 bh 值。即第 163 行前应该增加判断“if (bh->b_next)”。该错误到 0.96 版后
    // 才被纠正。
157         bh->b_prev = NULL;
158         bh->b_next = NULL;
159         if (!bh->b_dev)
160                 return;
161         bh->b_next = hash(bh->b_dev,bh->b_blocknr);
162         hash(bh->b_dev,bh->b_blocknr) = bh;
163         bh->b_next->b_prev = bh;          // 此句前应添加“if (bh->b_next)”判断。
164 }
165
    //// 利用 hash 表在高速缓冲中寻找给定设备和指定块号的缓冲区块。
    // 如果找到则返回缓冲区块的指针，否则返回 NULL。
166 static struct buffer_head * find_buffer(int dev, int block)
167 {
168         struct buffer_head * tmp;
169
    // 搜索 hash 表，寻找指定设备号和块号的缓冲块。
170         for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
171                 if (tmp->b_dev==dev && tmp->b_blocknr==block)
```

```
172                         return tmp;
173         return NULL;
174 }
175
176 /*
177  * Why like this, I hear you say... The reason is race-conditions.
178  * As we don't lock buffers (unless we are readint them, that is),
179  * something might happen to it while we sleep (ie a read-error
180  * will force it bad). This shouldn't really happen currently, but
181  * the code is ready.
182  */
    /*
     * 代码为什么会是这样子的？我听见你问... 原因是竞争条件。由于我们没有对
     * 缓冲块上锁（除非我们正在读取它们中的数据），那么当我们（进程）睡眠时
     * 缓冲块可能会发生一些问题（例如一个读错误将导致该缓冲块出错）。目前
     * 这种情况实际上是不会发生的，但处理的代码已经准备好了。
     */
    //// 利用 hash 表在高速缓冲区中寻找指定的缓冲块。若找到则对该缓冲块上锁并返回块头指针。
183 struct buffer_head * get_hash_table(int dev, int block)
184 {
185         struct buffer_head * bh;
186
187         for (;;) {
    // 在高速缓冲中寻找给定设备和指定块的缓冲区块，如果没有找到则返回 NULL，退出。
188                 if (!(bh=find_buffer(dev,block)))
189                         return NULL;
    // 对该缓冲块增加引用计数，并等待该缓冲块解锁（如果已被上锁）。由于经过了睡眠状态，
    // 因此有必要再验证该缓冲块的正确性，并返回缓冲块头指针。
190                 bh->b_count++;
191                 wait_on_buffer(bh);
192                 if (bh->b_dev == dev && bh->b_blocknr == block)
193                         return bh;
    // 如果在睡眠时该缓冲块所属的设备号或块号发生了改变，则撤消对它的引用计数，重新寻找。
194                 bh->b_count--;
195         }
196 }
197
198 /*
199  * Ok, this is getblk, and it isn't very clear, again to hinder
200  * race-conditions. Most of the code is seldom used, (ie repeating),
201  * so it should be much more efficient than it looks.
202  *
203  * The algoritm is changed: hopefully better, and an elusive bug removed.
204  */
    /*
     * OK，下面是 getblk 函数，该函数的逻辑并不是很清晰，同样也是因为要考虑
     * 竞争条件问题。其中大部分代码很少用到，(例如重复操作语句)，因此它应该
     * 比看上去的样子有效得多。
     *
     * 算法已经作了改变：希望能更好，而且一个难以琢磨的错误已经去除。
     */
    // 下面宏用于同时判断缓冲区的修改标志和锁定标志，并且定义修改标志的权重要比锁定标志
    // 大。
```

```
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
//// 取高速缓冲中指定的缓冲块。
// 检查指定（设备号和块号）的缓冲区是否已经在高速缓冲中。如果指定块已经在高速缓冲中，
// 则返回对应缓冲区头指针退出；如果不在，就需要在高速缓冲中设置一个对应设备号和块号的
// 新项。返回相应缓冲区头指针。
struct buffer_head * getblk(int dev,int block)
{
        struct buffer_head * tmp, * bh;

repeat:
// 搜索 hash 表，如果指定块已经在高速缓冲中，则返回对应缓冲区头指针，退出。
        if (bh = get_hash_table(dev,block))
                return bh;
// 扫描空闲数据块链表，寻找空闲缓冲区。
// 首先让 tmp 指向空闲链表的第一个空闲缓冲区头。
        tmp = free_list;
        do {
// 如果该缓冲区正被使用（引用计数不等于 0），则继续扫描下一项。对于 b_count=0 的块，
// 即高速缓冲中当前没有引用的块不一定就是干净的（b_dirt=0）或没有锁定的（b_lock=0）。
// 因此，我们还是需要继续下面的判断和选择。例如当一个任务改写过一块内容后就释放了，
// 于是该块 b_count = 0，但 b_lock 不等于 0；当一个任务执行 breada()预读几个块时，只要
// ll_rw_block()命令发出后，它就会递减 b_count；但此时实际上硬盘访问操作可能还在进行，
// 因此此时 b_lock=1，但 b_count=0。
                if (tmp->b_count)
                        continue;
// 如果缓冲头指针 bh 为空，或者 tmp 所指缓冲头的标志(修改、锁定)权重小于 bh 头标志的权
// 重，则让 bh 指向 tmp 缓冲块头。 如果该 tmp 缓冲块头表明缓冲块既没有修改也没有锁定标
// 志置位，则说明已为指定设备上的块取得对应的高速缓冲块，则退出循环。否则我们就继续
// 执行本循环，看看能否找到一个 BADNESS()最小的缓冲快。
                if (!bh || BADNESS(tmp)<BADNESS(bh)) {
                        bh = tmp;
                        if (!BADNESS(tmp))
                                break;
                }
/* and repeat until we find something good */ /* 重复操作直到找到适合的缓冲块 */
        } while ((tmp = tmp->b_next_free) != free_list);
// 如果循环检查发现所有缓冲块都正在被使用（所有缓冲块的头部引用计数都>0）中，则睡眠
// 等待有空闲缓冲块可用。当有空闲缓冲块可用时本进程会被明确地唤醒。然后我们就跳转到
// 函数开始处重新查找空闲缓冲块。
        if (!bh) {
                sleep_on(&buffer_wait);
                goto repeat;                    // 跳转至 210 行。
        }
// 执行到这里，说明我们已经找到了一个比较适合的空闲缓冲块了。于是先等待该缓冲区解锁
//（如果已被上锁的话）。如果在我们睡眠阶段该缓冲区又被其他任务使用的话，只好重复上述
// 寻找过程。
        wait_on_buffer(bh);
        if (bh->b_count)                        // 又被占用？？
                goto repeat;
// 如果该缓冲区已被修改，则将数据写盘，并再次等待缓冲区解锁。同样地，若该缓冲区又被
// 其他任务使用的话，只好再重复上述寻找过程。
        while (bh->b_dirt) {
                sync_dev(bh->b_dev);
```

```
233                 wait_on_buffer(bh);
234                 if (bh->b_count)                // 又被占用？？
235                         goto repeat;
236         }
237 /* NOTE!! While we slept waiting for this block, somebody else might */
238 /* already have added "this" block to the cache. check it */
    /* 注意！！当进程为了等待该缓冲块而睡眠时，其他进程可能已经将该缓冲块 */
     * 加入进高速缓冲中，所以我们也要对此进行检查。*/
    // 在高速缓冲 hash 表中检查指定设备和块的缓冲块是否乘我们睡眠之即已经被加入进去。如果
    // 是的话，就再次重复上述寻找过程。
239         if (find_buffer(dev,block))
240                 goto repeat;
241 /* OK, FINALLY we know that this buffer is the only one of it's kind, */
242 /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
    /* OK，最终我们知道该缓冲块是指定参数的唯一一块，而且目前还没有被占用 */
    /* (b_count=0)，也未被上锁(b_lock=0)，并且是干净的（未被修改的）*/
    // 于是让我们占用此缓冲块。置引用计数为 1，复位修改标志和有效(更新)标志。
243         bh->b_count=1;
244         bh->b_dirt=0;
245         bh->b_uptodate=0;
    // 从 hash 队列和空闲块链表中移出该缓冲区头，让该缓冲区用于指定设备和其上的指定块。
    // 然后根据此新的设备号和块号重新插入空闲链表和 hash 队列新位置处。并最终返回缓冲
    // 头指针。
246         remove_from_queues(bh);
247         bh->b_dev=dev;
248         bh->b_blocknr=block;
249         insert_into_queues(bh);
250         return bh;
251 }
252
    //// 释放指定缓冲块。
    // 等待该缓冲块解锁。然后引用计数递减 1，并明确地唤醒等待空闲缓冲块的进程。
253 void brelse(struct buffer_head * buf)
254 {
255         if (!buf)               // 如果缓冲头指针无效则返回。
256                 return;
257         wait_on_buffer(buf);
258         if (!(buf->b_count--))
259                 panic("Trying to free free buffer");
260         wake_up(&buffer_wait);
261 }
262
263 /*
264  * bread() reads a specified block and returns the buffer that contains
265  * it. It returns NULL if the block was unreadable.
266  */
    /*
     * 从设备上读取指定的数据块并返回含有数据的缓冲区。如果指定的块不存在
     * 则返回 NULL。
     */
    //// 从设备上读取数据块。
    // 该函数根据指定的设备号 dev 和数据块号 block，首先在高速缓冲区中申请一块缓冲块。
    // 如果该缓冲块中已经包含有有效的数据就直接返回该缓冲块指针，否则就从设备中读取
```

```
    // 指定的数据块到该缓冲块中并返回缓冲块指针。
267 struct buffer_head * bread(int dev,int block)
268 {
269         struct buffer_head * bh;
270
    // 在高速缓冲区中申请一块缓冲块。如果返回值是 NULL，则表示内核出错，停机。然后我们
    // 判断其中是否已有可用数据。 如果该缓冲块中数据是有效的（已更新的）可以直接使用，
    // 则返回。
271         if (!(bh=getblk(dev,block)))
272                 panic("bread: getblk returned NULL\n");
273         if (bh->b_uptodate)
274                 return bh;
    // 否则我们就调用底层块设备读写 ll_rw_block()函数，产生读设备块请求。然后等待指定
    // 数据块被读入，并等待缓冲区解锁。在睡眠醒来之后，如果该缓冲区已更新，则返回缓冲
    // 区头指针，退出。否则表明读设备操作失败，于是释放该缓冲区，返回 NULL，退出。
275         ll_rw_block(READ,bh);
276         wait_on_buffer(bh);
277         if (bh->b_uptodate)
278                 return bh;
279         brelse(bh);
280         return NULL;
281 }
282
    //// 复制内存块。
    // 从 from 地址复制一块（1024 字节）数据到 to 位置。
283 #define COPYBLK(from,to) \
284 __asm__("cld\n\t" \
285         "rep\n\t" \
286         "movsl\n\t" \
287         ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
288         :"cx","di","si")
289
290 /*
291  * bread_page reads four buffers into memory at the desired address. It's
292  * a function of its own, as there is some speed to be got by reading them
293  * all at the same time, not waiting for one to be read, and then another
294  * etc.
295  */
    /*
     * bread_page 一次读四个缓冲块数据读到内存指定的地址处。它是一个完整的函数，
     * 因为同时读取四块可以获得速度上的好处，不用等着读一块，再读一块了。
     */
    //// 读设备上一个页面（4 个缓冲块）的内容到指定内存地址处。
    // 参数 address 是保存页面数据的地址；dev 是指定的设备号；b[4]是含有 4 个设备数据块号
    // 的数组。该函数仅用于 mm/memory.c 文件的 do_no_page()函数中（第 386 行）。
296 void bread_page(unsigned long address,int dev,int b[4])
297 {
298         struct buffer_head * bh[4];
299         int i;
300
    // 该函数循环执行 4 次，根据放在数组 b[]中的 4 个块号从设备 dev 中读取一页内容放到指定
    // 内存位置 address 处。 对于参数 b[i]给出的有效块号，函数首先从高速缓冲中取指定设备
    // 和块号的缓冲块。如果缓冲块中数据无效（未更新）则产生读设备请求从设备上读取相应数
```

```
    // 据块。对于 b[i]无效的块号则不用去理它了。因此本函数其实可以根据指定的 b[]中的块号
    // 随意读取 1—4 个数据块。
301         for (i=0 ; i<4 ; i++)
302                 if (b[i]) {                     // 若块号有效。
303                         if (bh[i] = getblk(dev,b[i]))
304                                 if (!bh[i]->b_uptodate)
305                                         ll_rw_block(READ,bh[i]);
306                 } else
307                         bh[i] = NULL;
    // 随后将 4 个缓冲块上的内容顺序复制到指定地址处。在进行复制（使用）缓冲块之前我们
    // 先要睡眠等待缓冲块解锁（若被上锁的话）。另外，因为可能睡眠过了，所以我们还需要
    // 在复制之前再检查一下缓冲块中的数据是否是有效的。复制完后我们还需要释放缓冲块。
308         for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
309                 if (bh[i]) {
310                         wait_on_buffer(bh[i]);   // 等待缓冲块解锁(若被上锁的话)。
311                         if (bh[i]->b_uptodate)   // 若缓冲块中数据有效的话则复制。
312                                 COPYBLK((unsigned long) bh[i]->b_data,address);
313                         brelse(bh[i]);           // 释放该缓冲区。
314                 }
315 }
316
317 /*
318  * Ok, breada can be used as bread, but additionally to mark other
319  * blocks for reading as well. End the argument list with a negative
320  * number.
321  */
    /*
     * OK，breada 可以象 bread 一样使用，但会另外预读一些块。该函数参数列表
     * 需要使用一个负数来表明参数列表的结束。
     */
    //// 从指定设备读取指定的一些块。
    // 函数参数个数可变，是一系列指定的块号。成功时返回第 1 块的缓冲块头指针，否则返回
    // NULL。
322 struct buffer_head * breada(int dev,int first, ...)
323 {
324         va_list args;
325         struct buffer_head * bh, *tmp;
326
    // 首先取可变参数表中第 1 个参数（块号）。接着从高速缓冲区中取指定设备和块号的缓冲
    // 块。如果该缓冲块数据无效（更新标志未置位），则发出读设备数据块请求。
327         va_start(args,first);
328         if (!(bh=getblk(dev,first)))
329                 panic("bread: getblk returned NULL\n");
330         if (!bh->b_uptodate)
331                 ll_rw_block(READ,bh);
    // 然后顺序取可变参数表中其他预读块号，并作与上面同样处理，但不引用。注意，336 行上
    // 有一个 bug。其中的 bh 应该是 tmp。这个 bug 直到在 0.96 版的内核代码中才被纠正过来。
    // 另外，因为这里是预读随后的数据块，只需读进高速缓冲区但并不马上就使用，所以第 337
    // 行语句需要将其引用计数递减释放掉该块（因为 getblk()函数会增加缓冲块引用计数值）。
332         while ((first=va_arg(args,int))>=0) {
333                 tmp=getblk(dev,first);
334                 if (tmp) {
335                         if (!tmp->b_uptodate)
```

```
336                                 ll_rw_block(READA,bh);      // bh 应该是 tmp。
337                         tmp->b_count--;                     // 暂时释放掉该预读块。
338                 }
339         }
    // 此时可变参数表中所有参数处理完毕。于是等待第 1 个缓冲区解锁（如果已被上锁）。在等
    // 待退出之后如果缓冲区中数据仍然有效，则返回缓冲区头指针退出。否则释放该缓冲区返回
    // NULL，退出。
340         va_end(args);
341         wait_on_buffer(bh);
342         if (bh->b_uptodate)
343                 return bh;
344         brelse(bh);
345         return (NULL);
346 }
347
    //// 缓冲区初始化函数。
    // 参数 buffer_end 是缓冲区内存末端。对于具有 16MB 内存的系统，缓冲区末端被设置为 4MB。
    // 对于有 8MB 内存的系统，缓冲区末端被设置为 2MB。该函数从缓冲区开始位置 start_buffer
    // 处和缓冲区末端 buffer_end 处分别同时设置（初始化）缓冲块头结构和对应的数据块。直到
    // 缓冲区中所有内存被分配完毕。参见程序列表前面的示意图。
348 void buffer_init(long buffer_end)
349 {
350         struct buffer_head * h = start_buffer;
351         void * b;
352         int i;
353
    // 首先根据参数提供的缓冲区高端位置确定实际缓冲区高端位置 b。如果缓冲区高端等于 1Mb，
    // 则因为从 640KB - 1MB 被显示内存和 BIOS 占用，所以实际可用缓冲区内存高端位置应该是
    // 640KB。否则缓冲区内存高端一定大于 1MB。
354         if (buffer_end == 1<<20)
355                 b = (void *) (640*1024);
356         else
357                 b = (void *) buffer_end;
    // 这段代码用于初始化缓冲区，建立空闲缓冲块循环链表，并获取系统中缓冲块数目。操作的
    // 过程是从缓冲区高端开始划分 1KB 大小的缓冲块，与此同时在缓冲区低端建立描述该缓冲块
    // 的结构 buffer_head，并将这些 buffer_head 组成双向链表。
    // h 是指向缓冲头结构的指针，而 h+1 是指向内存地址连续的下一个缓冲头地址，也可以说是
    // 指向 h 缓冲头的末端外。为了保证有足够长度的内存来存储一个缓冲头结构，需要 b 所指向
    // 的内存块地址 >= h 缓冲头的末端，即要求 >= h+1。
358         while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
359                 h->b_dev = 0;                   // 使用该缓冲块的设备号。
360                 h->b_dirt = 0;                  // 脏标志，即缓冲块修改标志。
361                 h->b_count = 0;                 // 缓冲块引用计数。
362                 h->b_lock = 0;                  // 缓冲块锁定标志。
363                 h->b_uptodate = 0;              // 缓冲块更新标志（或称数据有效标志）。
364                 h->b_wait = NULL;               // 指向等待该缓冲块解锁的进程。
365                 h->b_next = NULL;               // 指向具有相同 hash 值的下一个缓冲头。
366                 h->b_prev = NULL;               // 指向具有相同 hash 值的前一个缓冲头。
367                 h->b_data = (char *) b;         // 指向对应缓冲块数据块（1024 字节）。
368                 h->b_prev_free = h-1;           // 指向链表中前一项。
369                 h->b_next_free = h+1;           // 指向链表中下一项。
370                 h++;                            // h 指向下一新缓冲头位置。
371                 NR_BUFFERS++;                   // 缓冲区块数累加。
```

```
372                 if (b == (void *) 0x100000)     // 若 b 递减到等于 1MB，则跳过 384KB，
373                         b = (void *) 0xA0000;   // 让 b 指向地址 0xA0000(640KB)处。
374         }
375         h--;                                    // 让 h 指向最后一个有效缓冲块头。
376         free_list = start_buffer;               // 让空闲链表头指向头一个缓冲块。
377         free_list->b_prev_free = h;  // 链表头的 b_prev_free 指向前一项（即最后一项）。
378         h->b_next_free = free_list;             // h 的下一项指针指向第一项，形成一个环链。
    // 最后初始化 hash 表（哈希表、散列表），置表中所有指针为 NULL。
379         for (i=0;i<NR_HASH;i++)
380                 hash_table[i]=NULL;
381 }
382
```