程序 9-3 linux/kernel/blk_drv/ll_rw_blk.c

```
 1 /*
 2  *  linux/kernel/blk_dev/ll_rw.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 /*
 8  * This handles all read/write requests to block devices
 9  */
10 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
11 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
15 #include "blk.h"          // 块设备头文件。定义请求数据结构、块设备数据结构和宏等信息。
16
17 /*
18  * The request-struct contains all necessary data
19  * to load a nr of sectors into memory
20  */
   /*
    * 请求结构中含有加载 nr 个扇区数据到内存中去的所有必须的信息。
    */
   // 请求项数组队列。共有 NR_REQUEST = 32 个请求项。
21 struct request request[NR_REQUEST];
22
23 /*
24  * used to wait on when there are no free requests
25  */
   /*
    * 是用于在请求数组没有空闲项时进程的临时等待处。
    */
26 struct task_struct * wait_for_request = NULL;
27
28 /* blk_dev_struct is:
29  *      do_request-address
30  *      next-request
31  */
   /* blk_dev_struct 块设备结构是：（参见文件 kernel/blk_drv/blk.h，第 45 行）
    *      do_request-address      // 对应主设备号的请求处理程序指针。
    *      current-request         // 该设备的下一个请求。
    */
   // 块设备数组。该数组使用主设备号作为索引。实际内容将在各块设备驱动程序初始化时填入。
   // 例如，硬盘驱动程序初始化时（hd.c，343 行），第一条语句即用于设置 blk_dev[3]的内容。
32 struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
33         { NULL, NULL },         /* no_dev */     // 0 - 无设备。
34         { NULL, NULL },         /* dev mem */    // 1 - 内存。
35         { NULL, NULL },         /* dev fd */     // 2 - 软驱设备。
36         { NULL, NULL },         /* dev hd */     // 3 - 硬盘设备。
37         { NULL, NULL },         /* dev ttyx */   // 4 - ttyx 设备。
38         { NULL, NULL },         /* dev tty */    // 5 - tty 设备。
39         { NULL, NULL }          /* dev lp */     // 6 - lp 打印机设备。
```

```
40 };
41
42 /*
43  * blk_size contains the size of all block-devices:
44  *
45  * blk_size[MAJOR][MINOR]
46  *
47  * if (!blk_size[MAJOR]) then no minor size checking is done.
48  */
   /*
    * blk_size 数组含有所有块设备的大小（块总数）：
    * blk_size[MAJOR][MINOR]
    * 如果 (!blk_size[MAJOR])，则不必检测子设备的块总数。
    */
   // 设备数据块总数指针数组。每个指针项指向指定主设备号的总块数数组。该总块数数组每一
   // 项对应子设备号确定的一个子设备上所拥有的数据块总数（1 块大小 = 1KB）。
49 int * blk_size[NR_BLK_DEV] = { NULL, NULL, };
50
   // 锁定指定缓冲块。
   // 如果指定的缓冲块已经被其他任务锁定，则使自己睡眠（不可中断地等待），直到被执行解
   // 锁缓冲块的任务明确地唤醒。
51 static inline void lock_buffer(struct buffer_head * bh)
52 {
53         cli();                          // 清中断许可。
54         while (bh->b_lock)              // 如果缓冲区已被锁定则睡眠，直到缓冲区解锁。
55                 sleep_on(&bh->b_wait);
56         bh->b_lock=1;                   // 立刻锁定该缓冲区。
57         sti();                          // 开中断。
58 }
59
   // 释放（解锁）锁定的缓冲区。
   // 该函数与 blk.h 文件中的同名函数完全一样。
60 static inline void unlock_buffer(struct buffer_head * bh)
61 {
62         if (!bh->b_lock)                // 如果该缓冲区没有被锁定，则打印出错信息。
63                 printk("ll_rw_block.c: buffer not locked\n\r");
64         bh->b_lock = 0;                 // 清锁定标志。
65         wake_up(&bh->b_wait);           // 唤醒等待该缓冲区的任务。
66 }
67
68 /*
69  * add-request adds a request to the linked list.
70  * It disables interrupts so that it can muck with the
71  * request-lists in peace.
72  *
73  * Note that swapping requests always go before other requests,
74  * and are done in the order they appear.
75  */
   /*
    * add-request()向链表中加入一项请求项。它会关闭中断，
    * 这样就能安全地处理请求链表了。
    *
    * 注意，交换请求总是在其他请求之前操作，并且以它们出
```

```
 * 现的顺序完成。
 */
//// 向链表中加入请求项。
// 参数 dev 是指定块设备结构指针，该结构中有处理请求项函数指针和当前正在请求项指针；
// req 是已设置好内容的请求项结构指针。
// 本函数把已经设置好的请求项 req 添加到指定设备的请求项链表中。如果该设备的当前请求
// 请求项指针为空，则可以设置 req 为当前请求项并立刻调用设备请求项处理函数。否则就把
// req 请求项插入到该请求项链表中。
76 static void add_request(struct blk_dev_struct * dev, struct request * req)
77 {
78         struct request * tmp;
79
// 首先再进一步对参数提供的请求项的指针和标志作初始设置。置空请求项中的下一请求项指
// 针，关中断并清除请求项相关缓冲区脏标志。
80         req->next = NULL;
81         cli();                                  // 关中断。
82         if (req->bh)
83                 req->bh->b_dirt = 0;            // 清缓冲区“脏”标志。
// 然后查看指定设备是否有当前请求项，即查看设备是否正忙。如果指定设备 dev 当前请求项
// （current_request）子段为空，则表示目前该设备没有请求项，本次是第 1 个请求项，也是
// 唯一的一个。因此可将块设备当前请求指针直接指向该请求项，并立刻执行相应设备的请求
// 函数。
84         if (!(tmp = dev->current_request)) {
85                 dev->current_request = req;
86                 sti();                          // 开中断。
87                 (dev->request_fn)();            // 执行请求函数，对于硬盘是 do_hd_request()。
88                 return;
89         }
// 如果目前该设备已经有当前请求项在处理，则首先利用电梯算法搜索最佳插入位置，然后将
// 当前请求项插入到请求链表中。在搜索过程中，如果判断出欲插入请求项的缓冲块头指针空，
// 即没有缓冲块，那么就需要找一个项，其已经有可用的缓冲块。因此若当前插入位置（tmp
// 之后）处的空闲项缓冲块头指针不空，就选择这个位置。于是退出循环并把请求项插入此处。
// 最后开中断并退出函数。电梯算法的作用是让磁盘磁头的移动距离最小，从而改善（减少）
// 硬盘访问时间。
// 下面 for 循环中 if 语句用于把 req 所指请求项与请求队列（链表）中已有的请求项作比较，
// 找出 req 插入该队列的正确位置顺序。然后中断循环，并把 req 插入到该队列正确位置处。
90         for ( ; tmp->next ; tmp=tmp->next) {
91                 if (!req->bh)
92                         if (tmp->next->bh)
93                                 break;
94                         else
95                                 continue;
96                 if ((IN_ORDER(tmp,req) ||
97                     !IN_ORDER(tmp,tmp->next)) &&
98                     IN_ORDER(req,tmp->next))
99                         break;
100        }
101        req->next=tmp->next;
102        tmp->next=req;
103        sti();
104 }
105
//// 创建请求项并插入请求队列中。
```

```
    // 参数 major 是主设备号；rw 是指定命令；bh 是存放数据的缓冲区头指针。
106 static void make_request(int major,int rw, struct buffer_head * bh)
107 {
108         struct request * req;
109         int rw_ahead;
110
111 /* WRITEA/READA is special case - it is not really needed, so if the */
112 /* buffer is locked, we just forget about it, else it's a normal read */
    /* WRITEA/READA 是一种特殊情况 - 它们并非必要，所以如果缓冲区已经上锁，*/
    /* 我们就不用管它，否则的话它只是一个一般的读操作。 */
    // 这里'READ'和'WRITE'后面的'A'字符代表英文单词 Ahead，表示提前预读/写数据块的意思。
    // 该函数首先对命令 READA/WRITEA 的情况进行一些处理。对于这两个命令，当指定的缓冲区
    // 正在使用而已被上锁时，就放弃预读/写请求。否则就作为普通的 READ/WRITE 命令进行操作。
    // 另外，如果参数给出的命令既不是 READ 也不是 WRITE，则表示内核程序有错，显示出错信
    // 息并停机。注意，在修改命令之前这里已为参数是否是预读/写命令设置了标志 rw_ahead。
113         if (rw_ahead = (rw == READA || rw == WRITEA)) {
114                 if (bh->b_lock)
115                         return;
116                 if (rw == READA)
117                         rw = READ;
118                 else
119                         rw = WRITE;
120         }
121         if (rw!=READ && rw!=WRITE)
122                 panic("Bad block dev command, must be R/W/RA/WA");
123         lock_buffer(bh);
124         if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
125                 unlock_buffer(bh);
126                 return;
127         }
128 repeat:
129 /* we don't allow the write-requests to fill up the queue completely:
130  * we want some room for reads: they take precedence. The last third
131  * of the requests are only for reads.
132  */
    /* 我们不能让队列中全都是写请求项：我们需要为读请求保留一些空间：读操作
     * 是优先的。请求队列的后三分之一空间仅用于读请求项。
     */
    // 好，现在我们必须为本函数生成并添加读/写请求项了。首先我们需要在请求数组中寻找到
    // 一个空闲项（槽）来存放新请求项。搜索过程从请求数组末端开始。根据上述要求，对于读
    // 命令请求，我们直接从队列末尾开始搜索，而对于写请求就只能从队列 2/3 处向队列头处搜
    // 索空项填入。于是我们开始从后向前搜索，当请求结构 request 的设备字段 dev 值 = -1 时，
    // 表示该项未被占用（空闲）。如果没有一项是空闲的（此时请求项数组指针已经搜索越过头
    // 部），则查看此次请求是否是提前读/写（READA 或 WRITEA），如果是则放弃此次请求操作。
    // 否则让本次请求操作先睡眠（以等待请求队列腾出空项），过一会再来搜索请求队列。
133         if (rw == READ)
134                 req = request+NR_REQUEST;           // 对于读请求，将指针指向队列尾部。
135         else
136                 req = request+((NR_REQUEST*2)/3);   // 对于写请求，指针指向队列 2/3 处。
137 /* find an empty request */                          /* 搜索一个空请求项 */
138         while (--req >= request)
139                 if (req->dev<0)
140                         break;
```

```
141 /* if none found, sleep on new requests: check for rw_ahead */
    /* 如果没有找到空闲项，则让该次新请求操作睡眠：需检查是否提前读/写 */
142         if (req < request) {                    // 如果已搜索到头（队列无空项），
143                 if (rw_ahead) {                 // 则若是提前读/写请求，则退出。
144                         unlock_buffer(bh);
145                         return;
146                 }
147                 sleep_on(&wait_for_request);    // 否则就睡眠，过会再查看请求队列。
148                 goto repeat;                    // 跳转 110 行。
149         }
150 /* fill up the request-info, and add it to the queue */
    /* 向空闲请求项中填写请求信息，并将其加入队列中 */
    // OK，程序执行到这里表示已找到一个空闲请求项。 于是我们在设置好的新请求项后就调用
    // add_request()把它添加到请求队列中，立马退出。请求结构请参见 blk_drv/blk.h，23 行。
    // req->sector 是读写操作的起始扇区号，req->buffer 是请求项存放数据的缓冲区。
151         req->dev = bh->b_dev;               // 设备号。
152         req->cmd = rw;                      // 命令(READ/WRITE)。
153         req->errors=0;                      // 操作时产生的错误次数。
154         req->sector = bh->b_blocknr<<1;     // 起始扇区。块号转换成扇区号(1 块=2 扇区)。
155         req->nr_sectors = 2;                // 本请求项需要读写的扇区数。
156         req->buffer = bh->b_data;           // 请求项缓冲区指针指向需读写的数据缓冲区。
157         req->waiting = NULL;                // 任务等待操作执行完成的地方。
158         req->bh = bh;                       // 缓冲块头指针。
159         req->next = NULL;                   // 指向下一请求项。
160         add_request(major+blk_dev,req);     // 将请求项加入队列中(blk_dev[major],req)。
161 }
162
    //// 低级页面读写函数（Low Level Read Write Page）。
    // 以页面（4K）为单位访问块设备数据，即每次读/写 8 个扇区。参见下面 ll_rw_blk()函数。
163 void ll_rw_page(int rw, int dev, int page, char * buffer)
164 {
165         struct request * req;
166         unsigned int major = MAJOR(dev);
167
    // 首先对函数参数的合法性进行检测。如果设备主设备号不存在或者该设备的请求操作函数不
    // 存在，则显示出错信息，并返回。如果参数给出的命令既不是 READ 也不是 WRITE，则表示
    // 内核程序有错，显示出错信息并停机。
168         if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
169                 printk("Trying to read nonexistent block-device\n\r");
170                 return;
171         }
172         if (rw!=READ && rw!=WRITE)
173                 panic("Bad block dev command, must be R/W");
    // 在参数检测操作完成后，我们现在需要为本次操作建立请求项。首先我们需要在请求数组中
    // 寻找到一个空闲项（槽）来存放新请求项。搜索过程从请求数组末端开始。于是我们开始从
    // 后向前搜索，当请求结构 request 的设备字段 dev 值 <0 时，表示该项未被占用（空闲）。
    // 如果没有一项是空闲的（此时请求项数组指针已经搜索越过头部），则让本次请求操作先睡
    // 眠（以等待请求队列腾出空项），过一会再来搜索请求队列。
174 repeat:
175         req = request+NR_REQUEST;               // 将指针指向队列尾部。
176         while (--req >= request)
177                 if (req->dev<0)
178                         break;
```

```
179         if (req < request) {
180                 sleep_on(&wait_for_request);        // 睡眠，过会再查看请求队列。
181                 goto repeat;                        // 跳转到 174 行去重新搜索。
182         }
183 /* fill up the request-info, and add it to the queue */
    /* 向空闲请求项中填写请求信息，并将其加入队列中 */
    // OK，程序执行到这里表示已找到一个空闲请求项。 于是我们设置好新请求项，把当前进程
    // 置为不可中断睡眠中断后，就去调用 add_request()把它添加到请求队列中，然后直接调用
    // 调度函数让当前进程睡眠等待页面从交换设备中读入。这里不象 make_request()函数那样
    // 直接退出函数而调用了 schedule()，是因为 make_request()函数仅读 2 个扇区数据。而这
    // 里需要对交换设备读/写 8 个扇区，需要花较长的时间。因此当前进程肯定需要等待而睡眠。
    // 因此这里直接就让进程去睡眠了，省得在程序其他地方还要进行这些判断操作。
184         req->dev = dev;                             // 设备号。
185         req->cmd = rw;                              // 命令(READ/WRITE)。
186         req->errors = 0;                            // 读写操作错误计数。
187         req->sector = page<<3;                      // 起始读写扇区。
188         req->nr_sectors = 8;                        // 读写扇区数。
189         req->buffer = buffer;                       // 数据缓冲区。
190         req->waiting = current;                     // 当前进程进入该请求等待队列。
191         req->bh = NULL;                             // 无缓冲块头指针（不用高速缓冲）。
192         req->next = NULL;                           // 下一个请求项指针。
193         current->state = TASK_UNINTERRUPTIBLE;      // 置为不可中断状态。
194         add_request(major+blk_dev,req);             // 将请求项加入队列中。
195         schedule();
196 }
197
    //// 低级数据块读写函数（Low Level Read Write Block）。
    // 该函数是块设备驱动程序与系统其他部分的接口函数。通常在 fs/buffer.c 程序中被调用。
    // 主要功能是创建块设备读写请求项并插入到指定块设备请求队列中。实际的读写操作则是
    // 由设备的 request_fn()函数完成。对于硬盘操作，该函数是 do_hd_request()；对于软盘
    // 操作该函数是 do_fd_request()；对于虚拟盘则是 do_rd_request()。 另外，在调用该函
    // 数之前，调用者需要首先把读/写块设备的信息保存在缓冲块头结构中，如设备号、块号。
    // 参数：rw - READ、READA、WRITE 或 WRITEA 是命令；bh - 数据缓冲块头指针。
198 void ll_rw_block(int rw, struct buffer_head * bh)
199 {
200         unsigned int major;                 // 主设备号（对于硬盘是 3）。
201
    // 如果设备主设备号不存在或者该设备的请求操作函数不存在，则显示出错信息，并返回。
    // 否则创建请求项并插入请求队列。
202         if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
203         !(blk_dev[major].request_fn)) {
204                 printk("Trying to read nonexistent block-device\n\r");
205                 return;
206         }
207         make_request(major,rw,bh);
208 }
209
    //// 块设备初始化函数，由初始化程序 main.c 调用。
    // 初始化请求数组，将所有请求项置为空闲项(dev = -1)。有 32 项(NR_REQUEST = 32)。
210 void blk_dev_init(void)
211 {
212         int i;
213
```

```
214         for (i=0 ; i<NR_REQUEST ; i++) {
215                 request[i].dev = -1;
216                 request[i].next = NULL;
217         }
218 }
219
```