```
1  /*
2   *  linux/kernel/hd.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * This is the low-level hd interrupt support. It traverses the
9   * request-list, using interrupts to jump between functions. As
10  * all the functions are called within interrupts, we may not
11  * sleep. Special care is recommended.
12  *
13  *  modified by Drew Eckhardt to check nr of hd's from the CMOS.
14  */
    /*
     * 本程序是底层硬盘中断辅助程序。主要用于扫描请求项队列，使用中断
     * 在函数之间跳转。由于所有的函数都是在中断里调用的，所以这些函数
     * 不可以睡眠。请特别注意。
     *
     * 由 Drew Eckhardt 修改，利用 CMOS 信息检测硬盘数。
     */
15
16 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）选项。
17 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
18 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file、m_inode）等。
19 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
20 #include <linux/hdreg.h>  // 硬盘参数头文件。定义硬盘寄存器端口、状态码、分区表等信息。
21 #include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等的汇编宏。
22 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
23 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
24
    // 定义硬盘主设备号符号常数。在驱动程序中，主设备号必须在包含 blk.h 文件之前被定义。
    // 因为 blk.h 文件中要用到这个符号常数值来确定一些列其他相关符号常数和宏。
25 #define MAJOR_NR 3        // 硬盘主设备号是 3。
26 #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。
27
    // 读 CMOS 参数宏函数。
    // 这段宏读取 CMOS 中硬盘信息。outb_p、inb_p 是 include/asm/io.h 中定义的端口输入输出宏。
    // 与 init/main.c 中读取 CMOS 时钟信息的宏完全一样。
28 #define CMOS_READ(addr) ({ \
29 outb_p(0x80|addr,0x70); \          // 0x70 是写端口号，0x80|addr 是要读的 CMOS 内存地址。
30 inb_p(0x71); \                     // 0x71 是读端口号。
31 })
32
33 /* Max read/write errors/sector */
    /* 每扇区读/写操作允许的最多出错次数 */
34 #define MAX_ERRORS      7         // 读/写一个扇区时允许的最多出错次数。
35 #define MAX_HD          2         // 系统支持的最多硬盘数。
36
    // 重新校正处理函数。
    // 复位操作时在硬盘中断处理程序中调用的重新校正函数(311 行)。
37 static void recal_intr(void);
```

```
   // 读写硬盘失败处理调用函数。
   // 结束本次请求项处理或者设置复位标志要求执行复位硬盘控制器操作后再重试（242 行）。
38 static void bad_rw_intr(void);
39
   // 重新校正标志。当设置了该标志，程序中会调用 recal_intr()以将磁头移动到 0 柱面。
40 static int recalibrate = 0;
  // 复位标志。当发生读写错误时会设置该标志并调用相关复位函数，以复位硬盘和控制器。
41 static int reset = 0;
42
43 /*
44  *  This struct defines the HD's and their types.
45  */
   /* 下面结构定义了硬盘参数及类型 */
   // 硬盘信息结构（Harddisk information struct）。
   // 各字段分别是磁头数、每磁道扇区数、柱面数、写前预补偿柱面号、磁头着陆区柱面号、
   // 控制字节。它们的含义请参见程序列表后的说明。
46 struct hd_i_struct {
47         int head,sect,cyl,wpcom,lzone,ctl;
48         };

   // 如果已经在 include/linux/config.h 配置文件中定义了符号常数 HD_TYPE，就取其中定义
   // 好的参数作为硬盘信息数组 hd_info[]中的数据。否则先默认都设为 0 值，在 setup()函数
   // 中会重新进行设置。
49 #ifdef HD_TYPE
50 struct hd_i_struct hd_info[] = { HD_TYPE };                    // 硬盘信息数组。
51 #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))  // 计算硬盘个数。
52 #else
53 struct hd_i_struct hd_info[] = { {0,0,0,0,0,0},{0,0,0,0,0,0} };
54 static int NR_HD = 0;
55 #endif
56
   // 定义硬盘分区结构。给出每个分区从硬盘 0 道开始算起的物理起始扇区号和分区扇区总数。
   // 其中 5 的倍数处的项（例如 hd[0]和 hd[5]等）代表整个硬盘的参数。
57 static struct hd_struct {
58         long start_sect;                // 分区在硬盘中的起始物理（绝对）扇区。
59         long nr_sects;                  // 分区中扇区总数。
60 } hd[5*MAX_HD]={{0,0},};
61
   // 硬盘每个分区数据块总数数组。
62 static int hd_sizes[5*MAX_HD] = {0, };
63
   // 读端口嵌入汇编宏。读端口 port，共读 nr 字，保存在 buf 中。
64 #define port_read(port,buf,nr) \
65 __asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr):"cx","di")
66
   // 写端口嵌入汇编宏。写端口 port，共写 nr 字，从 buf 中取数据。
67 #define port_write(port,buf,nr) \
68 __asm__("cld;rep;outsw"::"d" (port),"S" (buf),"c" (nr):"cx","si")
69

70 extern void hd_interrupt(void);          // 硬盘中断过程（sys_call.s，235 行）。
71 extern void rd_load(void);               // 虚拟盘创建加载函数（ramdisk.c，71 行）。
72
```

```
73 /* This may be used only once, enforced by 'static int callable' */
   /* 下面该函数只在初始化时被调用一次。用静态变量 callable 作为可调用标志。*/
   // 系统设置函数。
   // 函数参数 BIOS 是由初始化程序 init/main.c 中 init 子程序设置为指向硬盘参数表结构的指针。
   // 该硬盘参数表结构包含 2 个硬盘参数表的内容（共 32 字节），是从内存 0x90080 处复制而来。
   // 0x90080 处的硬盘参数表是由 setup.s 程序利用 ROM BIOS 功能取得。硬盘参数表信息参见程序
   // 列表后的说明。 本函数主要功能是读取 CMOS 和硬盘参数表信息，用于设置硬盘分区结构 hd，
   // 并尝试加载 RAM 虚拟盘和根文件系统。
74 int sys_setup(void * BIOS)
75 {
76         static int callable = 1;                // 限制本函数只能被调用 1 次的标志。
77         int i,drive;
78         unsigned char cmos_disks;
79         struct partition *p;
80         struct buffer_head * bh;
81
   // 首先设置 callable 标志，使得本函数只能被调用 1 次。然后设置硬盘信息数组 hd_info[]。
   // 如果在 include/linux/config.h 文件中已定义了符号常数 HD_TYPE，那么 hd_info[]数组
   // 已经在前面第 49 行上设置好了。否则就需要读取 boot/setup.s 程序存放在内存 0x90080 处
   // 开始的硬盘参数表。setup.s 程序在内存此处连续存放着一到两个硬盘参数表。
82         if (!callable)
83                 return -1;
84         callable = 0;
85 #ifndef HD_TYPE                                 // 如果没有定义 HD_TYPE，则读取。
86         for (drive=0 ; drive<2 ; drive++) {
87                 hd_info[drive].cyl = *(unsigned short *) BIOS;      // 柱面数。
88                 hd_info[drive].head = *(unsigned char *) (2+BIOS);  // 磁头数。
89                 hd_info[drive].wpcom = *(unsigned short *) (5+BIOS); // 写前预补偿柱面号。
90                 hd_info[drive].ctl = *(unsigned char *) (8+BIOS);    // 控制字节。
91                 hd_info[drive].lzone = *(unsigned short *) (12+BIOS);// 磁头着陆区柱面号。
92                 hd_info[drive].sect = *(unsigned char *) (14+BIOS);  // 每磁道扇区数。
93                 BIOS += 16;             // 每个硬盘参数表长 16 字节，这里 BIOS 指向下一表。
94         }
   // setup.s 程序在取 BIOS 硬盘参数表信息时，如果系统中只有 1 个硬盘，就会将对应第 2 个
   // 硬盘的 16 字节全部清零。因此这里只要判断第 2 个硬盘柱面数是否为 0 就可以知道是否有
   // 第 2 个硬盘了。
95         if (hd_info[1].cyl)
96                 NR_HD=2;                // 硬盘数置为 2。
97         else
98                 NR_HD=1;
99 #endif
   // 到这里，硬盘信息数组 hd_info[]已经设置好，并且确定了系统含有的硬盘数 NR_HD。现在
   // 开始设置硬盘分区结构数组 hd[]。该数组的项 0 和项 5 分别表示两个硬盘的整体参数，而
   // 项 1—4 和 6—9 分别表示两个硬盘的 4 个分区的参数。 因此这里仅设置表示硬盘整体信息
   // 的两项（项 0 和 5）。
100        for (i=0 ; i<NR_HD ; i++) {
101                hd[i*5].start_sect = 0;                         // 硬盘起始扇区号。
102                hd[i*5].nr_sects = hd_info[i].head*
103                                hd_info[i].sect*hd_info[i].cyl;  // 硬盘总扇区数。
104        }
105
106        /*
107                We querry CMOS about hard disks : it could be that
```

```
108                 we have a SCSI/ESDI/etc controller that is BIOS
109                 compatable with ST-506, and thus showing up in our
110                 BIOS table, but not register compatable, and therefore
111                 not present in CMOS.
112
113                 Furthurmore, we will assume that our ST-506 drives
114                 <if any> are the primary drives in the system, and
115                 the ones reflected as drive 1 or 2.
116
117                 The first drive is stored in the high nibble of CMOS
118                 byte 0x12, the second in the low nibble.  This will be
119                 either a 4 bit drive type or 0xf indicating use byte 0x19
120                 for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.
121
122                 Needless to say, a non-zero value means we have
123                 an AT controller hard disk for that drive.
124
125
126         */
          /*
                  我们对 CMOS 有关硬盘的信息有些怀疑：可能会出现这样的情况，
                  我们有一块 SCSI/ESDI/等的控制器，它是以 ST-506 方式与 BIOS
                  相兼容的，因而会出现在我们的 BIOS 参数表中，但却又不是寄存
                  器兼容的，因此这些参数在 CMOS 中又不存在。

                  另外，我们假设 ST-506 驱动器（如果有的话）是系统中的基本驱
                  动器，也即以驱动器 1 或 2 出现的驱动器。

                  第 1 个驱动器参数存放在 CMOS 字节 0x12 的高半字节中，第 2 个
                  存放在低半字节中。该 4 位字节信息可以是驱动器类型，也可能
                  仅是 0xf。0xf 表示使用 CMOS 中 0x19 字节作为驱动器 1 的 8 位
                  类型字节，使用 CMOS 中 0x1A 字节作为驱动器 2 的类型字节。

                  总之，一个非零值意味着硬盘是一个 AT 控制器兼容硬盘。
           */
127
    // 这里根据上述原理，下面代码用来检测硬盘到底是不是 AT 控制器兼容的。有关 CMOS 信息
    // 请参见第 4 章中 4.2.3.1 节。这里从 CMOS 偏移地址 0x12 处读出硬盘类型字节。如果低半
    // 字节值（存放着第 2 个硬盘类型值）不为 0，则表示系统有两硬盘，否则表示系统只有 1
    // 个硬盘。如果 0x12 处读出的值为 0，则表示系统中没有 AT 兼容硬盘。
128         if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
129                 if (cmos_disks & 0x0f)
130                         NR_HD = 2;
131                 else
132                         NR_HD = 1;
133         else
134                 NR_HD = 0;
    // 若 NR_HD = 0，则两个硬盘都不是 AT 控制器兼容的，两个硬盘数据结构全清零。
    // 若 NR_HD = 1，则将第 2 个硬盘的参数清零。
135         for (i = NR_HD ; i < 2 ; i++) {
136                 hd[i*5].start_sect = 0;
137                 hd[i*5].nr_sects = 0;
138         }
```

```
    // 好，到此为止我们已经真正确定了系统中所含的硬盘个数 NR_HD。现在我们来读取每个硬盘
    // 上第 1 个扇区中的分区表信息，用来设置分区结构数组 hd[] 中硬盘各分区的信息。首先利
    // 用读块函数 bread()读硬盘第 1 个数据块（fs/buffer.c，第 267 行），第 1 个参数（0x300、
    // 0x305 ）分别是两个硬盘的设备号，第 2 个参数（0）是所需读取的块号。若读操作成功，
    // 则数据会被存放在缓冲块 bh 的数据区中。若缓冲块头指针 bh 为 0，则说明读操作失败，则
    // 显示出错信息并停机。否则我们根据硬盘第 1 个扇区最后两个字节应该是 0xAA55 来判断扇
    // 区中数据的有效性，从而可以知道扇区中位于偏移 0x1BE 开始处的分区表是否有效。若有效
    // 则将硬盘分区表信息放入硬盘分区结构数组 hd[]中。最后释放 bh 缓冲区。
139         for (drive=0 ; drive<NR_HD ; drive++) {
140                 if (!(bh = bread(0x300 + drive*5,0))) {  // 0x300、0x305 是设备号。
141                         printk("Unable to read partition table of drive %d\n\r",
142                                 drive);
143                         panic("");
144                 }
145                 if (bh->b_data[510] != 0x55 || (unsigned char)
146                     bh->b_data[511] != 0xAA) {            // 判断硬盘标志 0xAA55。
147                         printk("Bad partition table on drive %d\n\r",drive);
148                         panic("");
149                 }
150                 p = 0x1BE + (void *)bh->b_data;     // 分区表位于第 1 扇区 0x1BE 处。
151                 for (i=1;i<5;i++,p++) {
152                         hd[i+5*drive].start_sect = p->start_sect;
153                         hd[i+5*drive].nr_sects = p->nr_sects;
154                 }
155                 brelse(bh);                 // 释放为存放硬盘数据块而申请的缓冲区。
156         }
    // 现在再对每个分区中的数据块总数进行统计，并保存在硬盘分区总数据块数组 hd_sizes[]中。
    // 然后让设备数据块总数指针数组的本设备项指向该数组。
157         for (i=0 ; i<5*MAX_HD ; i++)
158                 hd_sizes[i] = hd[i].nr_sects>>1 ;
159         blk_size[MAJOR_NR] = hd_sizes;
    // 现在总算完成设置硬盘分区结构数组 hd[]的任务。如果确实有硬盘存在并且已读入其分区
    // 表，则显示“分区表正常”信息。然后尝试在系统内存虚拟盘中加载启动盘中包含的根文
    // 件系统映像（blk_drv/ramdisk.c，第 71 行）。即在系统设置有虚拟盘的情况下判断启动盘
    // 上是否还含有根文件系统的映像数据。如果有（此时该启动盘称为集成盘）则尝试把该映像
    // 加载并存放到虚拟盘中，然后把此时的根文件系统设备号 ROOT_DEV 修改成虚拟盘的设备号。
    // 接着再对交换设备进行初始化。最后安装根文件系统。
160         if (NR_HD)
161                 printk("Partition table%s ok.\n\r",(NR_HD>1)?"s":"");
162         rd_load();                          // blk_drv/ramdisk.c，第 71 行。
163         init_swapping();                    // mm/swap.c，第 199 行。
164         mount_root();                       // fs/super.c，第 241 行。
165         return (0);
166 }
167
    //// 判断并循环等待硬盘控制器就绪。
    // 读硬盘控制器状态寄存器端口 HD_STATUS(0x1f7)，循环检测其中的驱动器就绪比特位（位 6）
    // 是否被置位并且控制器忙位（位 7）是否被复位。 如果返回值 retries 为 0，则表示等待控制
    // 器空闲的时间已经超时而发生错误，若返回值不为 0 则说明在等待（循环）时间期限内控制器
    // 回到空闲状态，OK！
    // 实际上，我们仅需检测状态寄存器忙位（位 7）是否为 1 来判断控制器是否处于忙状态，驱动
    // 器是否就绪（即位 6 是否为 1）与控制器的状态无关。因此我们可以把第 172 行语句改写成：
    // “while (--retries && (inb_p(HD_STATUS)&0x80));”另外，由于现在的 PC 机速度都很快，
```

```
    // 因此我们可以把等待的循环次数再加大一些，例如再增加 10 倍！
168 static int controller_ready(void)
169 {
170         int retries = 100000;
171
172         while (--retries && (inb_p(HD_STATUS)&0xc0)!=0x40);
173         return (retries);                       // 返回等待循环次数。
174 }
175
    //// 检测硬盘执行命令后的状态。（win 表示温切斯特硬盘的缩写）
    // 读取状态寄存器中的命令执行结果状态。 返回 0 表示正常；1 表示出错。如果执行命令错，
    // 则需要再读错误寄存器 HD_ERROR（0x1f1）。
176 static int win_result(void)
177 {
178         int i=inb_p(HD_STATUS);             // 取状态信息。
179
180         if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
181                 == (READY_STAT | SEEK_STAT))
182                 return(0); /* ok */
183         if (i&1) i=inb(HD_ERROR);           // 若 ERR_STAT 置位，则读取错误寄存器。
184         return (1);
185 }
186
    //// 向硬盘控制器发送命令块。
    // 参数：drive - 硬盘号(0-1)；nsect - 读写扇区数；sect  - 起始扇区；
    //       head  - 磁头号；      cyl   - 柱面号；      cmd   - 命令码（见控制器命令列表）；
    //       intr_addr() - 硬盘中断处理程序中将调用的 C 处理函数指针。
    // 该函数在硬盘控制器就绪之后，先设置全局指针变量 do_hd 为硬盘中断处理程序中将调用的
    // C 处理函数指针。然后再发送硬盘控制字节和 7 字节的参数命令块。
    // 该函数在硬盘控制器就绪之后，先设置全局函数指针变量 do_hd 指向硬盘中断处理程序中将会
    // 调用的 C 处理函数，然后再发送硬盘控制字节和 7 字节的参数命令块。硬盘中断处理程序的代
    // 码位于 kernel/sys_call.s 程序第 235 行处。
    // 第 191 行定义 1 个寄存器变量__res。该变量将被保存在 1 个寄存器中，以便于快速访问。
    // 如果想指定寄存器（如 eax），则我们可以把该句写成“register char __res asm("ax");”。
187 static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
188                 unsigned int head,unsigned int cyl,unsigned int cmd,
189                 void (*intr_addr)(void))
190 {
191         register int port asm("dx");    // 定义局部寄存器变量并放在指定寄存器 dx 中。
192
    // 首先对参数进行有效性检查。如果驱动器号大于 1（只能是 0、1）或者磁头号大于 15，则程
    // 序不支持，停机。否则就判断并循环等待驱动器就绪。如果等待一段时间后仍未就绪则表示
    // 硬盘控制器出错，也停机。
193         if (drive>1 || head>15)
194                 panic("Trying to write bad sector");
195         if (!controller_ready())
196                 panic("HD controller not ready");
    // 接着我们设置硬盘中断发生时将调用的 C 函数指针 do_hd（该函数指针定义在 blk.h 文件的
    // 第 56--109 行之间，请特别留意其中的第 83 行和 100 行）。然后在向硬盘控制器发送参数
    // 和命令之前，规定要先向控制器命令端口（0x3f6）发送一指定硬盘的控制字节，以建立相
    // 应的硬盘控制方式。该控制字节即是硬盘信息结构数组中的 ctl 字段。然后向控制器端口
    // 0x1f1 - 0x1f7 发送 7 字节的参数命令块。
197         SET_INTR(intr_addr);                    // do_hd = intr_addr 在中断中被调用。
```

```
198         outb_p(hd_info[drive].ctl,HD_CMD);      // 向控制寄存器输出控制字节。
199         port=HD_DATA;                           // 置 dx 为数据寄存器端口(0x1f0)。
200         outb_p(hd_info[drive].wpcom>>2,++port); // 参数：写预补偿柱面号(需除 4)。
201         outb_p(nsect,++port);                   // 参数：读/写扇区总数。
202         outb_p(sect,++port);                    // 参数：起始扇区。
203         outb_p(cyl,++port);                     // 参数：柱面号低 8 位。
204         outb_p(cyl>>8,++port);                  // 参数：柱面号高 8 位。
205         outb_p(0xA0|(drive<<4)|head,++port);    // 参数：驱动器号+磁头号。
206         outb(cmd,++port);                       // 命令：硬盘控制命令。
207 }
208
    //// 等待硬盘就绪。
    // 该函数循环等待主状态控制器忙标志位复位。若仅有就绪或寻道结束标志置位，则表示硬盘
    // 就绪，成功返回 0。若经过一段时间仍为忙，则返回 1。
209 static int drive_busy(void)
210 {
211         unsigned int i;
212         unsigned char c;
213
    // 循环读取控制器的主状态寄存器 HD_STATUS，等待就绪标志位置位并且忙位复位。然后检测
    // 其中忙位、就绪位和寻道结束位。若仅有就绪或寻道结束标志置位，则表示硬盘就绪，返回
    // 0。否则表示等待超时。于是警告显示信息。并返回 1。
214         for (i = 0; i < 50000; i++) {
215                 c = inb_p(HD_STATUS);                   // 取主控制器状态字节。
216                 c &= (BUSY_STAT | READY_STAT | SEEK_STAT);
217                 if (c == (READY_STAT | SEEK_STAT))
218                         return 0;
219         }
220         printk("HD controller times out\n\r");  // 等待超时，显示信息。并返回 1。
221         return(1);
222 }
223
    //// 诊断复位（重新校正）硬盘控制器。
    // 首先向控制寄存器端口（0x3f6）发送允许复位（4）控制字节。然后循环空操作等待一段时
    // 间让控制器执行复位操作。接着再向该端口发送正常的控制字节(不禁止重试、重读)，并等
    // 待硬盘就绪。若等待硬盘就绪超时，则显示警告信息。然后读取错误寄存器内容，若其不等
    // 于 1（表示无错误）则显示硬盘控制器复位失败信息。
224 static void reset_controller(void)
225 {
226         int     i;
227
228         outb(4,HD_CMD);                         // 向控制寄存器端口发送复位控制字节。
229         for(i = 0; i < 1000; i++) nop();        // 等待一段时间。
230         outb(hd_info[0].ctl & 0x0f ,HD_CMD);    // 发送正常控制字节(不禁止重试、重读)。
231         if (drive_busy())
232                 printk("HD-controller still busy\n\r");
233         if ((i = inb(HD_ERROR)) != 1)
234                 printk("HD-controller reset failed: %02x\n\r",i);
235 }
236
    //// 硬盘复位操作。
    // 首先复位（重新校正）硬盘控制器。然后发送硬盘控制器命令“建立驱动器参数”。在本
    // 命令引起的硬盘中断处理程序中又会调用本函数。此时该函数会根据执行该命令的结果判
```

```
    // 断是否要进行出错处理或是继续执行请求项处理操作。
237 static void reset_hd(void)
238 {
239         static int i;
240
    // 如果复位标志 reset 是置位的，则在把复位标志清零后，执行复位硬盘控制器操作。然后
    // 针对第 i 个硬盘向控制器发送“建立驱动器参数”命令。当控制器执行了该命令后，又会
    // 发出硬盘中断信号。此时本函数会被中断过程调用而再次执行。由于 reset 已经标志复位，
    // 因此会首先去执行 246 行开始的语句，判断命令执行是否正常。若还是发生错误就会调用
    // bad_rw_intr() 函数以统计出错次数并根据次确定是否在设置 reset 标志。如果又设置了
    // reset 标志则跳转到 repeat 重新执行本函数。若复位操作正常，则针对下一个硬盘发送
    // “建立驱动器参数”命令，并作上述同样处理。如果系统中 NR_HD 个硬盘都已经正常执行
    // 了发送的命令，则再次 do_hd_request()函数开始对请求项进行处理。
241 repeat:
242         if (reset) {
243                 reset = 0;
244                 i = -1;                         // 初始化当前硬盘号（静态变量）。
245                 reset_controller();
246         } else if (win_result()) {
247                 bad_rw_intr();
248                 if (reset)
249                         goto repeat;
250         }
251         i++;                                    // 处理下一个硬盘（第 1 个是 0）。
252         if (i < NR_HD) {
253                 hd_out(i,hd_info[i].sect,hd_info[i].sect,hd_info[i].head-1,
254                         hd_info[i].cyl,WIN_SPECIFY,&reset_hd);
255         } else
256                 do_hd_request();                // 执行请求项处理。
257 }
258
    //// 意外硬盘中断调用函数。
    // 发生意外硬盘中断时，硬盘中断处理程序中调用的默认 C 处理函数。在被调用函数指针为
    // NULL 时调用该函数。参见（kernel/sys_call.s，第 256 行）。该函数在显示警告信息后
    // 设置复位标志 reset，然后继续调用请求项函数 go_hd_request()并在其中执行复位处理
    // 操作。
259 void unexpected_hd_interrupt(void)
260 {
261         printk("Unexpected HD interrupt\n\r");
262         reset = 1;
263         do_hd_request();
264 }
265
    //// 读写硬盘失败处理调用函数。
    // 如果读扇区时的出错次数大于或等于 7 次时，则结束当前请求项并唤醒等待该请求的进程，
    // 而且对应缓冲区更新标志复位，表示数据没有更新。如果读写一扇区时的出错次数已经大于
    // 3 次，则要求执行复位硬盘控制器操作（设置复位标志）。
266 static void bad_rw_intr(void)
267 {
268         if (++CURRENT->errors >= MAX_ERRORS)
269                 end_request(0);
270         if (CURRENT->errors > MAX_ERRORS/2)
271                 reset = 1;
```

```
272 }
273
    //// 读操作中断调用函数。
    // 该函数将在硬盘读命令结束时引发的硬盘中断过程中被调用。
    // 在读命令执行后会产生硬盘中断信号，并执行硬盘中断处理程序，此时在硬盘中断处理程序
    // 中调用的 C 函数指针 do_hd 已经指向 read_intr()，因此会在一次读扇区操作完成（或出错）
    // 后就会执行该函数。
274 static void read_intr(void)
275 {
    // 该函数首先判断此次读命令操作是否出错。若命令结束后控制器还处于忙状态，或者命令
    // 执行错误，则处理硬盘操作失败问题，接着再次请求硬盘作复位处理并执行其他请求项。
    // 然后返回。每次读操作出错都会对当前请求项作出错次数累计，若出错次数不到最大允许
    // 出错次数的一半，则会先执行硬盘复位操作，然后再执行本次请求项处理。若出错次数已
    // 经大于等于最大允许出错次数 MAX_ERRORS（7 次），则结束本次请求项的处理而去处理队
    // 列中下一个请求项。
276         if (win_result()) {                 // 若控制器忙、读写错或命令执行错，
277                 bad_rw_intr();              // 则进行读写硬盘失败处理。
278                 do_hd_request();            // 再次请求硬盘作相应(复位)处理。
279                 return;
280         }
    // 如果读命令没有出错，则从数据寄存器端口把 1 个扇区的数据读到请求项的缓冲区中，并且
    // 递减请求项所需读取的扇区数值。若递减后不等于 0，表示本项请求还有数据没取完，于是
    // 再次置中断调用 C 函数指针 do_hd 为 read_intr()并直接返回，等待硬盘在读出另 1 个扇区
    // 数据后发出中断并再次调用本函数。注意：281 行语句中的 256 是指内存字，即 512 字节。
    // 注意 1：262 行再次置 do_hd 指针指向 read_intr()是因为硬盘中断处理程序每次调用 do_hd
    // 时都会将该函数指针置空。参见 sys_call.s 程序第 251—253 行。
281         port_read(HD_DATA,CURRENT->buffer,256);     // 读数据到请求结构缓冲区。
282         CURRENT->errors = 0;                // 清出错次数。
283         CURRENT->buffer += 512;             // 调整缓冲区指针，指向新的空区。
284         CURRENT->sector++;                  // 起始扇区号加 1，
285         if (--CURRENT->nr_sectors) {        // 如果所需读出的扇区数还没读完，则再
286                 SET_INTR(&read_intr);       // 置硬盘调用 C 函数指针为 read_intr()。
287                 return;
288         }
    // 执行到此，说明本次请求项的全部扇区数据已经读完，则调用 end_request()函数去处理请
    // 求项结束事宜。 最后再次调用 do_hd_request()，去处理其他硬盘请求项。执行其他硬盘
    // 请求操作。
289         end_request(1);                         // 数据已更新标志置位（1）。
290         do_hd_request();
291 }
292
    //// 写扇区中断调用函数。
    // 该函数将在硬盘写命令结束时引发的硬盘中断过程中被调用。函数功能与 read_intr()类似。
    // 在写命令执行后会产生硬盘中断信号，并执行硬盘中断处理程序，此时在硬盘中断处理程序
    // 中调用的 C 函数指针 do_hd 已经指向 write_intr()，因此会在一次写扇区操作完成（或出错）
    // 后就会执行该函数。
293 static void write_intr(void)
294 {
    // 该函数首先判断此次写命令操作是否出错。若命令结束后控制器还处于忙状态，或者命令
    // 执行错误，则处理硬盘操作失败问题，接着再次请求硬盘作复位处理并执行其他请求项。
    // 然后返回。在 bad_rw_intr() 函数中，每次操作出错都会对当前请求项作出错次数累计，
    // 若出错次数不到最大允许出错次数的一半，则会先执行硬盘复位操作，然后再执行本次请
    // 求项处理。若出错次数已经大于等于最大允许出错次数 MAX_ERRORS（7 次），则结束本次
```

```
    // 请求项的处理而去处理队列中下一个请求项。do_hd_request()中会根据当时具体的标志
    // 状态来判别是否需要先执行复位、重新校正等操作，然后再继续或处理下一个请求项。
295         if (win_result()) {                 // 如果硬盘控制器返回错误信息，
296                 bad_rw_intr();              // 则首先进行硬盘读写失败处理，
297                 do_hd_request();            // 再次请求硬盘作相应(复位)处理。
298                 return;
299         }
    // 此时说明本次写一扇区操作成功，因此将欲写扇区数减 1。若其不为 0，则说明还有扇区
    // 要写，于是把当前请求起始扇区号 +1，并调整请求项数据缓冲区指针指向下一块欲写的
    // 数据。然后再重置硬盘中断处理程序中调用的 C 函数指针 do_hd（指向本函数）。接着向
    // 控制器数据端口写入 512 字节数据，然后函数返回去等待控制器把这些数据写入硬盘后产
    // 生的中断。
300         if (--CURRENT->nr_sectors) {        // 若还有扇区要写，则
301                 CURRENT->sector++;          // 当前请求起始扇区号+1，
302                 CURRENT->buffer += 512;     // 调整请求缓冲区指针，
303                 SET_INTR(&write_intr);      // do_hd 置函数指针为 write_intr()。
304                 port_write(HD_DATA,CURRENT->buffer,256);  // 向数据端口写 256 字。
305                 return;
306         }
    // 若本次请求项的全部扇区数据已经写完，则调用 end_request()函数去处理请求项结束事宜。
    // 最后再次调用 do_hd_request()，去处理其他硬盘请求项。执行其他硬盘请求操作。
307         end_request(1);                     // 处理请求结束事宜（已设置更新标志）。
308         do_hd_request();                    // 执行其他硬盘请求操作。
309 }
310
    //// 硬盘重新校正（复位）中断调用函数。
    // 该函数会在硬盘执行重新校正操作而引发的硬盘中断中被调用。
    // 如果硬盘控制器返回错误信息，则函数首先进行硬盘读写失败处理，然后请求硬盘作相应
    // （复位）处理。 在 bad_rw_intr() 函数中，每次操作出错都会对当前请求项作出错次数
    // 累计，若出错次数不到最大允许出错次数的一半，则会先执行硬盘复位操作，然后再执行
    // 本次请求项处理。若出错次数已经大于等于最大允许出错次数 MAX_ERRORS（7 次），则结
    // 束本次请求项的处理而去处理队列中下一个请求项。do_hd_request() 中会根据当时具体
    // 的标志状态来判别是否需要先执行复位、重新校正等操作，然后再继续或处理下一请求项。
311 static void recal_intr(void)
312 {
313         if (win_result())                   // 若返回出错，则调用 bad_rw_intr()。
314                 bad_rw_intr();
315         do_hd_request();
316 }
317
    // 硬盘操作超时处理。
    // 本函数会在 do_timer()中（kernel/sched.c，第 340 行）被调用。在向硬盘控制器发送了
    // 一个命令后，若在经过了 hd_timeout 个系统滴答后控制器还没有发出一个硬盘中断信号，
    // 则说明控制器（或硬盘）操作超时。此时 do_timer()就会调用本函数设置复位标志 reset
    // 并调用 do_hd_request()执行复位处理。若在预定时间内（200 滴答）硬盘控制器发出了硬
    // 盘中断并开始执行硬盘中断处理程序，那么 ht_timeout 值就会在中断处理程序中被置 0。
    // 此时 do_timer()就会跳过本函数。
318 void hd_times_out(void)
319 {
    // 如果当前并没有请求项要处理（设备请求项指针为 NULL），则无超时可言，直接返回。否
    // 则先显示警告信息，然后判断当前请求项执行过程中发生的出错次数是否已经大于设定值
    // MAX_ERRORS（7）。如果是则以失败形式结束本次请求项的处理（不设置数据更新标志）。
    // 然后把中断过程中调用的 C 函数指针 do_hd 置空，并设置复位标志 reset，继而在请求项
```

```
    // 处理函数 do_hd_request()中去执行复位操作。
320         if (!CURRENT)
321                 return;
322         printk("HD timeout");
323         if (++CURRENT->errors >= MAX_ERRORS)
324                 end_request(0);
325         SET_INTR(NULL);                         // 令 do_hd = NULL,time_out=200。
326         reset = 1;                              // 设置复位标志。
327         do_hd_request();
328 }
329
    //// 执行硬盘读写请求操作。
    // 该函数根据设备当前请求项中的设备号和起始扇区号信息首先计算得到对应硬盘上的柱面号、
    // 当前磁道中扇区号、磁头号数据，然后再根据请求项中的命令（READ/WRITE）对硬盘发送相应
    // 读/写命令。 若控制器复位标志或硬盘重新校正标志已被置位，那么首先会去执行复位或重新
    // 校正操作。
    // 若请求项此时是块设备的第 1 个（原来设备空闲），则块设备当前请求项指针会直接指向该请
    // 求项（参见 ll_rw_blk.c，28 行），并会立刻调用本函数执行读写操作。否则在一个读写操作
    // 完成而引发的硬盘中断过程中，若还有请求项需要处理，则也会在硬盘中断过程中调用本函数。
    // 参见 kernel/sys_call.s，235 行。
330 void do_hd_request(void)
331 {
332         int i,r;
333         unsigned int block,dev;
334         unsigned int sec,head,cyl;
335         unsigned int nsect;
336
    // 函数首先检测请求项的合法性。若请求队列中已没有请求项则退出（参见 blk.h，127 行）。
    // 然后取设备号中的子设备号（见列表后对硬盘设备号的说明）以及设备当前请求项中的起始
    // 扇区号。子设备号即对应硬盘上各分区。如果子设备号不存在或者起始扇区大于该 分区扇
    // 区数-2，则结束该请求项，并跳转到标号 repeat 处（定义在 INIT_REQUEST 开始处）。因为
    // 一次要求读写一块数据（2 个扇区，即 1024 字节），所以请求的扇区号不能大于分区中最后
    // 倒数第二个扇区号。然后通过加上子设备号对应分区的起始扇区号，就把需要读写的块对应
    // 到整个硬盘的绝对扇区号 block 上。而子设备号被 5 整除即可得到对应的硬盘号。
337         INIT_REQUEST;
338         dev = MINOR(CURRENT->dev);
339         block = CURRENT->sector;                        // 请求的起始扇区。
340         if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
341                 end_request(0);
342                 goto repeat;                            // 该标号在 blk.h 最后面。
343         }
344         block += hd[dev].start_sect;
345         dev /= 5;                       // 此时 dev 代表硬盘号（硬盘 0 还是硬盘 1）。
    // 然后根据求得的绝对扇区号 block 和硬盘号 dev，我们就可以计算出对应硬盘中的磁道中扇
    // 区号（sec）、所在柱面号（cyl） 和磁头号（head）。 下面嵌入的汇编代码即用来根据硬
    // 盘信息结构中的每磁道扇区数和硬盘磁头数来计算这些数据。计算方法为：
    // 310--311 行代码初始时 eax 是扇区号 block，edx 中置 0。divl 指令把 edx:eax 组成的扇区
    // 号除以每磁道扇区数（hd_info[dev].sect），所得整数商值在 eax 中，余数在 edx 中。其
    // 中 eax 中是到指定位置的对应总磁道数（所有磁头面），edx 中是当前磁道上的扇区号。
    // 312--313 行代码初始时 eax 是计算出的对应总磁道数，edx 中置 0。divl 指令把 edx:eax
    // 的对应总磁道数除以硬盘总磁头数（hd_info[dev].head），在 eax 中得到的整除值是柱面
    // 号（cyl），edx 中得到的余数就是对应得当前磁头号（head）。
346         __asm__("divl %4":"=a" (block),"=d" (sec):"0" (block),"1" (0),
```

```
347                 "r" (hd_info[dev].sect));
348         __asm__("divl %4":"=a" (cyl),"=d" (head):"0" (block),"1" (0),
349                 "r" (hd_info[dev].head));
350         sec++;                                  // 对计算所得当前磁道扇区号进行调整。
351         nsect = CURRENT->nr_sectors;            // 欲读/写的扇区数。
    // 此时我们得到了欲读写的硬盘起始扇区 block 所对应的硬盘上柱面号（cyl）、在当前磁道
    // 上的扇区号（sec）、磁头号（head）以及欲读写的总扇区数（nsect）。 接着我们可以根
    // 据这些信息向硬盘控制器发送 I/O 操作信息了。 但在发送之前我们还需要先看看是否有复
    // 位控制器状态和重新校正硬盘的标志。通常在复位操作之后都需要重新校正硬盘磁头位置。
    // 若这些标志已被置位，则说明前面的硬盘操作可能出现了一些问题，或者现在是系统第一
    // 次硬盘读写操作等情况。 于是我们就需要重新复位硬盘或控制器并重新校正硬盘。

    // 如果此时复位标志 reset 是置位的，则需要执行复位操作。复位硬盘和控制器，并置硬盘
    // 需要重新校正标志，返回。reset_hd()将首先向硬盘控制器发送复位（重新校正）命令，
    // 然后发送硬盘控制器命令“建立驱动器参数”。
352         if (reset) {
353                 recalibrate = 1;                // 置需重新校正标志。
354                 reset_hd();
355                 return;
356         }
    // 如果此时重新校正标志（recalibrate）是置位的，则首先复位该标志，然后向硬盘控制
    // 器发送重新校正命令。该命令会执行寻道操作，让处于任何地方的磁头移动到 0 柱面。
357         if (recalibrate) {
358                 recalibrate = 0;
359                 hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
360                         WIN_RESTORE,&recal_intr);
361                 return;
362         }
    // 如果以上两个标志都没有置位，那么我们就可以开始向硬盘控制器发送真正的数据读/写
    // 操作命令了。如果当前请求是写扇区操作，则发送写命令，循环读取状态寄存器信息并判
    // 断请求服务标志 DRQ_STAT 是否置位。DRQ_STAT 是硬盘状态寄存器的请求服务位，表示驱
    // 动器已经准备好在主机和数据端口之间传输一个字或一个字节的数据。这方面的信息可参
    // 见程序前面的硬盘操作读/写时序图。如果请求服务 DRQ 置位则退出循环。 若等到循环结
    // 束也没有置位，则表示发送的要求写硬盘命令失败，于是跳转去处理出现的问题或继续执
    // 行下一个硬盘请求。否则我们就可以向硬盘控制器数据寄存器端口 HD_DATA 写入 1 个扇区
    // 的数据。
363         if (CURRENT->cmd == WRITE) {
364                 hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
365                 for(i=0 ; i<10000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
366                         /* nothing */ ;
367                 if (!r) {
368                         bad_rw_intr();
369                         goto repeat;                    // 该标号在 blk.h 文件最后面。
370                 }
371                 port_write(HD_DATA,CURRENT->buffer,256);
    // 如果当前请求是读硬盘数据，则向硬盘控制器发送读扇区命令。若命令无效则停机。
372         } else if (CURRENT->cmd == READ) {
373                 hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
374         } else
375                 panic("unknown hd-command");
376 }
377
    // 硬盘系统初始化。
```

```
    // 设置硬盘中断描述符，并允许硬盘控制器发送中断请求信号。
    // 该函数设置硬盘设备的请求项处理函数指针为 do_hd_request()，然后设置硬盘中断门描述
    // 符。hd_interrupt（kernel/sys_call.s，第 235 行）是其中断处理过程地址。 硬盘中断号
    // 为 int 0x2E（46），对应 8259A 芯片的中断请求信号 IRQ13。接着复位接联的主 8259A int2
    // 的屏蔽位，允许从片发出中断请求信号。再复位硬盘的中断请求屏蔽位（在从片上），允许
    // 硬盘控制器发送中断请求信号。中断描述符表 IDT 内中断门描述符设置宏 set_intr_gate()
    // 在 include/asm/system.h 中实现。
378 void hd_init(void)
379 {
380         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;      // do_hd_request()。
381         set_intr_gate(0x2E,&hd_interrupt);  // 设置中断门中处理函数指针。
382         outb_p(inb_p(0x21)&0xfb,0x21);      // 复位接联的主 8259A int2 的屏蔽位。
383         outb(inb_p(0xA1)&0xbf,0xA1);        // 复位硬盘中断请求屏蔽位（在从片上）。
384 }
385
```