```
1  /*
2   *  linux/kernel/blk_drv/ramdisk.c
3   *
4   *  Written by Theodore Ts'o, 12/2/91
5   */
   /* 由 Theodore Ts'o 编制，12/2/91
    */
   // Theodore Ts'o (Ted Ts'o)是 Linux 社区中的著名人物。Linux 在世界范围内的流行也有他很
   // 大的功劳。早在 Linux 操作系统刚问世时，他就怀着极大的热情为 Linux 的发展提供了电子邮
   // 件列表服务 maillist，并在北美地区最早设立了 Linux 的 ftp 服务器站点（tsx-11.mit.edu），
   // 而且至今仍为广大 Linux 用户提供服务。他对 Linux 作出的最大贡献之一是提出并实现了 ext2
   // 文件系统。该文件系统已成为 Linux 世界中事实上的文件系统标准。最近他又推出了 ext3 文件
   // 系统，大大提高了文件系统的稳定性、可恢复性和访问效率。作为对他的推崇，第 97 期（2002
   // 年 5 月）的 LinuxJournal 期刊将他作为了封面人物，并对他进行了采访。目前他为 IBM Linux
   // 技术中心工作，并从事着有关 LSB (Linux Standard Base)等方面的工作。（他的个人主页是：
   // http://thunk.org/tytso/)
6
7  #include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8
9  #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。
10 #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。
11 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file、m_inode）等。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原型定义。
13 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等嵌入式汇编宏。
14 #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。
15 #include <asm/memory.h>   // 内存拷贝头文件。含有 memcpy()嵌入式汇编宏函数。
16
   // 定义 RAM 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的其他常数符号和宏。
17 #define MAJOR_NR 1
18 #include "blk.h"
19
   // 虚拟盘在内存中的起始位置。该位置会在第 52 行上初始化函数 rd_init()中确定。参见内核
   // 初始化程序 init/main.c，第 124 行。'rd'是'ramdisk'的缩写。
20 char    *rd_start;                    // 虚拟盘在内存中的开始地址。
21 int     rd_length = 0;                // 虚拟盘所占内存大小（字节）。
22
   // 虚拟盘当前请求项操作函数。
   // 该函数的程序结构与硬盘的 do_hd_request()函数类似，参见 hd.c，294 行。在低级块设备
   // 接口函数 ll_rw_block()建立起虚拟盘（rd）的请求项并添加到 rd 的链表中之后，就会调
   // 用该函数对 rd 当前请求项进行处理。该函数首先计算当前请求项中指定的起始扇区对应虚
   // 拟盘所处内存的起始位置 addr 和要求的扇区数对应的字节长度值 len，然后根据请求项中
   // 的命令进行操作。若是写命令 WRITE，就把请求项所指缓冲区中的数据直接复制到内存位置
   // addr 处。若是读操作则反之。数据复制完成后即可直接调用 end_request() 对本次请求项
   // 作结束处理。然后跳转到函数开始处再去处理下一个请求项。若已没有请求项则退出。
23 void do_rd_request(void)
24 {
25         int     len;
26         char    *addr;
27
   // 首先检测请求项的合法性，若已没有请求项则退出（参见 blk.h，第 127 行）。然后计算请
```

```
    // 求项处理的虚拟盘中起始扇区在物理内存中对应的地址 addr 和占用的内存字节长度值 len。
    // 下句用于取得请求项中的起始扇区对应的内存起始位置和内存长度。其中 sector << 9 表示
    // sector * 512，换算成字节值。CURRENT 被定义为 (blk_dev[MAJOR_NR].current_request)。
28          INIT_REQUEST;
29          addr = rd_start + (CURRENT->sector << 9);
30          len = CURRENT->nr_sectors << 9;
    // 如果当前请求项中子设备号不为 1 或者对应内存起始位置大于虚拟盘末尾，则结束该请求项，
    // 并跳转到 repeat 处去处理下一个虚拟盘请求项。标号 repeat 定义在宏 INIT_REQUEST 内，
    // 位于宏的开始处，参见 blk.h 文件第 127 行。
31          if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
32                  end_request(0);
33                  goto repeat;
34          }
    // 然后进行实际的读写操作。如果是写命令（WRITE），则将请求项中缓冲区的内容复制到地址
    // addr 处，长度为 len 字节。如果是读命令（READ），则将 addr 开始的内存内容复制到请求项
    // 缓冲区中，长度为 len 字节。否则显示命令不存在，死机。
35          if (CURRENT-> cmd == WRITE) {
36                  (void ) memcpy(addr,
37                                  CURRENT->buffer,
38                                  len);
39          } else if (CURRENT->cmd == READ) {
40                  (void) memcpy(CURRENT->buffer,
41                                  addr,
42                                  len);
43          } else
44                  panic("unknown ramdisk-command");
    // 然后在请求项成功后处理，置更新标志。并继续处理本设备的下一请求项。
45          end_request(1);
46          goto repeat;
47  }
48
49  /*
50   * Returns amount of memory which needs to be reserved.
51   */
    /* 返回内存虚拟盘 ramdisk 所需的内存量 */
    // 虚拟盘初始化函数。
    // 该函数首先设置虚拟盘设备的请求项处理函数指针指向 do_rd_request()，然后确定虚拟盘
    // 在物理内存中的起始地址、占用字节长度值。并对整个虚拟盘区清零。最后返回盘区长度。
    // 当 linux/Makefile 文件中设置过 RAMDISK 值不为零时，表示系统中会创建 RAM 虚拟盘设备。
    // 在这种情况下的内核初始化过程中，本函数就会被调用（init/main.c，L151 行）。该函数
    // 的第 2 个参数 length 会被赋值成 RAMDISK * 1024，单位为字节。
52  long rd_init(long mem_start, int length)
53  {
54          int     i;
55          char    *cp;
56
57          blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;  // do_rd_request()。
58          rd_start = (char *) mem_start;                  // 对于 16MB 系统该值为 4MB。
59          rd_length = length;                             // 虚拟盘区域长度值。
60          cp = rd_start;
61          for (i=0; i < length; i++)                      // 盘区清零。
62                  *cp++ = '\0';
63          return(length);
```

```
64 }
65
66 /*
67  * If the root device is the ram disk, try to load it.
68  * In order to do this, the root device is originally set to the
69  * floppy, and we later change it to be ram disk.
70  */
   /*
    * 如果根文件系统设备(root device)是 ramdisk 的话，则尝试加载它。
    * root device 原先是指向软盘的，我们将它改成指向 ramdisk。
    */
   //// 尝试把根文件系统加载到虚拟盘中。
   // 该函数将在内核设置函数 setup()（hd.c，156 行）中被调用。另外，1 磁盘块 = 1024 字节。
   // 第 75 行上的变量 block=256 表示根文件系统映像文件被存储于 boot 盘第 256 磁盘块开始处。
71 void rd_load(void)
72 {
73         struct buffer_head *bh;         // 高速缓冲块头指针。
74         struct super_block      s;      // 文件超级块结构。
75         int             block = 256;    /* Start at block 256 */ /* 开始于 256 盘块 */
76         int             i = 1;
77         int             nblocks;        // 文件系统盘块总数。
78         char            *cp;            /* Move pointer */
79
   // 首先检查虚拟盘的有效性和完整性。如果 ramdisk 的长度为零，则退出。否则显示 ramdisk
   // 的大小以及内存起始位置。如果此时根文件设备不是软盘设备，则也退出。
80         if (!rd_length)
81                 return;
82         printk("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
83                 (int) rd_start);
84         if (MAJOR(ROOT_DEV) != 2)
85                 return;
   // 然后读根文件系统的基本参数。即读软盘块 256+1、256 和 256+2。这里 block+1 是指磁盘上
   // 的超级块。breada() 用于读取指定的数据块，并标出还需要读的块，然后返回含有数据块的
   // 缓冲区指针。如果返回 NULL，则表示数据块不可读（fs/buffer.c，322）。然后把缓冲区中
   // 的磁盘超级块（d_super_block 是磁盘超级块结构）复制到 s 变量中，并释放缓冲区。 接着
   // 我们开始对超级块的有效性进行判断。 如果超级块中文件系统魔数不对，则说明加载的数据
   // 块不是 MINIX 文件系统，于是退出。有关 MINIX 超级块的结构请参见文件系统一章内容。
86         bh = breada(ROOT_DEV,block+1,block,block+2,-1);
87         if (!bh) {
88                 printk("Disk error while looking for ramdisk!\n");
89                 return;
90         }
91         *((struct d_super_block *) &s) = *((struct d_super_block *) bh->b_data);
92         brelse(bh);
93         if (s.s_magic != SUPER_MAGIC)
94                 /* No ram disk image present, assume normal floppy boot */
                  /* 磁盘中没有 ramdisk 映像文件，退出去执行通常的软盘引导 */
95                 return;
   // 然后我们试图把整个根文件系统读入到内存虚拟盘区中。对于一个文件系统来说，其超级块
   // 结构的 s_nzones 字段中保存着总逻辑块数（或称为区段数）。一个逻辑块中含有的数据块
   // 数则由字段 s_log_zone_size 指定。因此文件系统中的数据块总数 nblocks 就等于 (逻辑块
   // 数 * 2^(每区段块数的次方))，即 nblocks = (s_nzones * 2^s_log_zone_size)。如果遇到
   // 文件系统中数据块总数大于内存虚拟盘所能容纳的块数的情况，则不能执行加载操作，而只
```

```
    // 能显示出错信息并返回。
 96         nblocks = s.s_nzones << s.s_log_zone_size;
 97         if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
 98                 printk("Ram disk image too big!  (%d blocks, %d avail)\n",
 99                         nblocks, rd_length >> BLOCK_SIZE_BITS);
100                 return;
101         }
    // 否则若虚拟盘能容纳得下文件系统总数据块数，则我们显示加载数据块信息，并让 cp 指向
    // 内存虚拟盘起始处，然后开始执行循环操作将磁盘上根文件系统映像文件加载到虚拟盘上。
    // 在操作过程中，如果一次需要加载的盘块数大于 2 块，我们就是用超前预读函数 breada()，
    // 否则就使用 bread()函数进行单块读取。若在读盘过程中出现 I/O 操作错误，就只能放弃加
    // 载过程返回。所读取的磁盘块会使用 memcpy()函数从高速缓冲区中复制到内存虚拟盘相应
    // 位置处，同时显示已加载的块数。显示字符串中的八进制数'\010'表示显示一个制表符。
102         printk("Loading %d bytes into ram disk... 0000k",
103                 nblocks << BLOCK_SIZE_BITS);
104         cp = rd_start;
105         while (nblocks) {
106                 if (nblocks > 2)                // 若读取块数多于 2 块则采用超前预读。
107                         bh = breada(ROOT_DEV, block, block+1, block+2, -1);
108                 else                            // 否则就单块读取。
109                         bh = bread(ROOT_DEV, block);
110                 if (!bh) {
111                         printk("I/O error on block %d, aborting load\n",
112                                 block);
113                         return;
114                 }
115                 (void) memcpy(cp, bh->b_data, BLOCK_SIZE);      // 复制到 cp 处。
116                 brelse(bh);
117                 printk("\010\010\010\010\010%4dk",i);          // 打印加载块计数值。
118                 cp += BLOCK_SIZE;                               // 虚拟盘指针前移。
119                 block++;
120                 nblocks--;
121                 i++;
122         }
    // 当 boot 盘中从 256 盘块开始的整个根文件系统加载完毕后，我们显示“done”，并把目前
    // 根文件设备号修改成虚拟盘的设备号 0x0101，最后返回。
123         printk("\010\010\010\010\010done \n");
124         ROOT_DEV=0x0101;
125 }
126
```