程序 12-9 linux/fs/file_dev.c

```
1  /*
2   *  linux/fs/file_dev.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
8  #include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
9
10 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据等。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #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。
13
14 #define MIN(a,b) (((a)<(b))?(a):(b))          // 取 a,b 中的最小值。
15 #define MAX(a,b) (((a)>(b))?(a):(b))          // 取 a,b 中的最大值。
16
   //// 文件读函数 - 根据 i 节点和文件结构，读取文件中数据。
   // 由 i 节点我们可以知道设备号，由 filp 结构可以知道文件中当前读写指针位置。buf 指定用
   // 户空间中缓冲区的位置，count 是需要读取的字节数。 返回值是实际读取的字节数，或出错
   // 号（小于 0）。
17 int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18 {
19         int left,chars,nr;
20         struct buffer_head * bh;
21
   // 首先判断参数的有效性。若需要读取的字节计数 count 小于等于零，则返回 0。若还需要读
   // 取的字节数不等于 0，就循环执行下面操作，直到数据全部读出或遇到问题。在读循环操作
   // 过程中，我们根据 i 节点和文件表结构信息，并利用 bmap() 得到包含文件当前读写位置的
   // 数据块在设备上对应的逻辑块号 nr。若 nr 不为 0，则从 i 节点指定的设备上读取该逻辑块。
   // 如果读操作失败则退出循环。若 nr 为 0，表示指定的数据块不存在，置缓冲块指针为 NULL。
   // (filp->f_pos)/BLOCK_SIZE 用于计算出文件当前指针所在数据块号。
22         if ((left=count)<=0)
23                 return 0;
24         while (left) {
25                 if (nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE)) { // inode.c 第 140 行。
26                         if (!(bh=bread(inode->i_dev,nr)))
27                                 break;
28                 } else
29                         bh = NULL;
   // 接着我们计算文件读写指针在数据块中的偏移值 nr，则在该数据块中我们希望读取的字节数
   // 为 (BLOCK_SIZE - nr)。然后和现在还需读取的字节数 left 作比较，其中小值即为本次操作
   // 需读取的字节数 chars。如果 (BLOCK_SIZE - nr) > left，则说明该块是需要读取的最后一
   // 块数据，反之则还需要读取下一块数据。之后调整读写文件指针。 指针前移此次将读取的字
   // 节数 chars。剩余字节计数 left 相应减去 chars。
30                 nr = filp->f_pos % BLOCK_SIZE;
31                 chars = MIN( BLOCK_SIZE-nr , left );
32                 filp->f_pos += chars;
33                 left -= chars;
   // 若上面从设备上读到了数据，则将 p 指向缓冲块中开始读取数据的位置，并且复制 chars 字节
   // 到用户缓冲区 buf 中。否则往用户缓冲区中填入 chars 个 0 值字节。
34                 if (bh) {
35                         char * p = nr + bh->b_data;
```

```
36                      while (chars-->0)
37                              put_fs_byte(*(p++),buf++);
38                      brelse(bh);
39              } else {
40                      while (chars-->0)
41                              put_fs_byte(0,buf++);
42              }
43      }
// 修改该 i 节点的访问时间为当前时间。返回读取的字节数，若读取字节数为 0，则返回出错号。
// CURRENT_TIME 是定义在 include/linux/sched.h 第 142 行上的宏，用于计算 UNIX 时间。即从
// 1970 年 1 月 1 日 0 时 0 秒开始，到当前的时间。单位是秒。
44      inode->i_atime = CURRENT_TIME;
45      return (count-left)?(count-left):-ERROR;
46 }
47
//// 文件写函数 - 根据 i 节点和文件结构信息，将用户数据写入文件中。
// 由 i 节点我们可以知道设备号，而由 file 结构可以知道文件中当前读写指针位置。buf 指定
// 用户态中缓冲区的位置，count 为需要写入的字节数。 返回值是实际写入的字节数，或出错
// 号（小于 0）。
48 int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
49 {
50      off_t pos;
51      int block,c;
52      struct buffer_head * bh;
53      char * p;
54      int i=0;
55
56 /*
57  * ok, append may not work when many processes are writing at the same time
58  * but so what. That way leads to madness anyway.
59  */
/*
 * ok，当许多进程同时写时，append 操作可能不行，但那又怎样。不管怎样那样做会
 * 导致混乱一团。
 */
// 首先确定数据写入文件的位置。如果是要向文件后添加数据，则将文件读写指针移到文件尾
// 部。否则就将在文件当前读写指针处写入。
60      if (filp->f_flags & O_APPEND)
61              pos = inode->i_size;
62      else
63              pos = filp->f_pos;
// 然后在已写入字节数 i（刚开始时为 0）小于指定写入字节数 count 时，循环执行以下操作。
// 在循环操作过程中，我们先取文件数据块号 （ pos/BLOCK_SIZE ）在设备上对应的逻辑块号
// block。如果对应的逻辑块不存在就创建一块。如果得到的逻辑块号 = 0，则表示创建失败，
// 于是退出循环。否则我们根据该逻辑块号读取设备上的相应逻辑块，若出错也退出循环。
64      while (i<count) {
65              if (!(block = create_block(inode,pos/BLOCK_SIZE)))
66                      break;
67              if (!(bh=bread(inode->i_dev,block)))
68                      break;
// 此时缓冲块指针 bh 正指向刚读入的文件数据块。现在再求出文件当前读写指针在该数据块中
// 的偏移值 c，并将指针 p 指向缓冲块中开始写入数据的位置，并置该缓冲块已修改标志。对于
// 块中当前指针，从开始读写位置到块末共可写入 c =(BLOCK_SIZE - c)个字节。若 c 大于剩余
```

```
    // 还需写入的字节数 (count - i)，则此次只需再写入 c = (count - i)个字节即可。
69                  c = pos % BLOCK_SIZE;
70                  p = c + bh->b_data;
71                  bh->b_dirt = 1;
72                  c = BLOCK_SIZE-c;
73                  if (c > count-i) c = count-i;
    // 在写入数据之前，我们先预先设置好下一次循环操作要读写文件中的位置。因此我们把 pos
    // 指针前移此次需写入的字节数。如果此时 pos 位置值超过了文件当前长度，则修改 i 节点中
    // 文件长度字段，并置 i 节点已修改标志。然后把此次要写入的字节数 c 累加到已写入字节计
    // 数值 i 中，供循环判断使用。接着从用户缓冲区 buf 中复制 c 个字节到高速缓冲块中 p 指向
    // 的开始位置处。复制完后就释放该缓冲块。
74                  pos += c;
75                  if (pos > inode->i_size) {
76                          inode->i_size = pos;
77                          inode->i_dirt = 1;
78                  }
79                  i += c;
80                  while (c-->0)
81                          *(p++) = get_fs_byte(buf++);
82                  brelse(bh);
83          }
    // 当数据已经全部写入文件或者在写操作过程中发生问题时就会退出循环。此时我们更改文件
    // 修改时间为当前时间，并调整文件读写指针。如果此次操作不是在文件尾添加数据，则把文
    // 件读写指针调整到当前读写位置 pos 处，并更改文件 i 节点的修改时间为当前时间。最后返
    // 回写入的字节数，若写入字节数为 0，则返回出错号-1。
84          inode->i_mtime = CURRENT_TIME;
85          if (!(filp->f_flags & O_APPEND)) {
86                  filp->f_pos = pos;
87                  inode->i_ctime = CURRENT_TIME;
88          }
89          return (i?i:-1);
90  }
91
```