```
1  /*
2   *  linux/fs/pipe.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <signal.h>        // 信号头文件。定义信号符号常量，信号结构及操作函数原型。
8  #include <errno.h>         // 错误号头文件。包含系统中各种出错号。
9  #include <termios.h>       // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
10
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/mm.h>      /* for get_free_page */     /* 使用其中的 get_free_page */
13 #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。
14 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
15
   //// 管道读操作函数。
   // 参数 inode 是管道对应的 i 节点，buf 是用户数据缓冲区指针，count 是读取的字节数。
16 int read_pipe(struct m_inode * inode, char * buf, int count)
17 {
18         int chars, size, read = 0;
19
   // 如果需要读取的字节计数 count 大于 0，我们就循环执行以下操作。在循环读操作过程中，
   // 若当前管道中没有数据（size=0），则唤醒等待该节点的进程，这通常是写管道进程。如果
   // 已没有写管道者，即 i 节点引用计数值小于 2，则返回已读字节数退出。如果当前收到有非
   // 阻塞信号，则立刻返回已读取字节数退出；若还没有收到任何数据，则返回重新启动系统
   // 调用号退出。否则就让进程在该管道上睡眠，用以等待信息的到来。宏 PIPE_SIZE 定义在
   // include/linux/fs.h 中。关于“重新启动系统调用”，请参见 kernel/signal.c 程序。
20         while (count>0) {
21                 while (!(size=PIPE_SIZE(*inode))) {      // 取管道中数据长度值。
22                         wake_up(& PIPE_WRITE_WAIT(*inode));
23                         if (inode->i_count != 2)    /* are there any writers? */
24                                 return read;
25                         if (current->signal & ~current->blocked)
26                                 return read?read:-ERESTARTSYS;
27                         interruptible_sleep_on(& PIPE_READ_WAIT(*inode));
28                 }
   // 此时说明管道（缓冲区）中有数据。于是我们取管道尾指针到缓冲区末端的字节数 chars。
   // 如果其大于还需要读取的字节数 count，则令其等于 count。如果 chars 大于当前管道中含
   // 有数据的长度 size，则令其等于 size。 然后把需读字节数 count 减去此次可读的字节数
   // chars，并累加已读字节数 read。
29                 chars = PAGE_SIZE-PIPE_TAIL(*inode);
30                 if (chars > count)
31                         chars = count;
32                 if (chars > size)
33                         chars = size;
34                 count -= chars;
35                 read += chars;
   // 再令 size 指向管道尾指针处，并调整当前管道尾指针（前移 chars 字节）。若尾指针超过
   // 管道末端则绕回。然后将管道中的数据复制到用户缓冲区中。对于管道 i 节点，其 i_size
   // 字段中是管道缓冲块指针。
36                 size = PIPE_TAIL(*inode);
37                 PIPE_TAIL(*inode) += chars;
```

```
38                  PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
39                  while (chars-->0)
40                          put_fs_byte(((char *)inode->i_size)[size++],buf++);
41          }
    // 当此次读管道操作结束，则唤醒等待该管道的进程，并返回读取的字节数。
42          wake_up(& PIPE_WRITE_WAIT(*inode));
43          return read;
44  }
45
    //// 管道写操作函数。
    // 参数 inode 是管道对应的 i 节点，buf 是数据缓冲区指针，count 是将写入管道的字节数。
46  int write_pipe(struct m_inode * inode, char * buf, int count)
47  {
48          int chars, size, written = 0;
49
    // 如果要写入的字节数 count 还大于 0，那么我们就循环执行以下操作。在循环操作过程中，
    // 如果当前管道中已经满了（空闲空间 size = 0），则唤醒等待该管道的进程，通常唤醒
    // 的是读管道进程。 如果已没有读管道者，即 i 节点引用计数值小于 2，则向当前进程发送
    // SIGPIPE 信号，并返回已写入的字节数退出；若写入 0 字节，则返回 -1。否则让当前进程
    // 在该管道上睡眠，以等待读管道进程来读取数据，从而让管道腾出空间。宏 PIPE_SIZE()、
    // PIPE_HEAD()等定义在文件 include/linux/fs.h 中。
50          while (count>0) {
51                  while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
52                          wake_up(& PIPE_READ_WAIT(*inode));
53                          if (inode->i_count != 2) { /* no readers */
54                                  current->signal |= (1<<(SIGPIPE-1));
55                                  return written?written:-1;
56                          }
57                          sleep_on(& PIPE_WRITE_WAIT(*inode));
58                  }
    // 程序执行到这里表示管道缓冲区中有可写空间 size。于是我们取管道头指针到缓冲区末端空
    // 间字节数 chars。写管道操作是从管道头指针处开始写的。如果 chars 大于还需要写入的字节
    // 数 count，则令其等于 count。 如果 chars 大于当前管道中空闲空间长度 size，则令其等于
    // size。然后把需要写入字节数 count 减去此次可写入的字节数 chars，并把写入字节数累加到
    // written 中。
59                  chars = PAGE_SIZE-PIPE_HEAD(*inode);
60                  if (chars > count)
61                          chars = count;
62                  if (chars > size)
63                          chars = size;
64                  count -= chars;
65                  written += chars;
    // 再令 size 指向管道数据头指针处，并调整当前管道数据头部指针（前移 chars 字节）。若头
    // 指针超过管道末端则绕回。然后从用户缓冲区复制 chars 个字节到管道头指针开始处。 对于
    // 管道 i 节点，其 i_size 字段中是管道缓冲块指针。
66                  size = PIPE_HEAD(*inode);
67                  PIPE_HEAD(*inode) += chars;
68                  PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
69                  while (chars-->0)
70                          ((char *)inode->i_size)[size++]=get_fs_byte(buf++);
71          }
    // 当此次写管道操作结束，则唤醒等待管道的进程，返回已写入的字节数，退出。
72          wake_up(& PIPE_READ_WAIT(*inode));
```

```
73          return written;
74  }
75
    //// 创建管道系统调用。
    // 在 fildes 所指的数组中创建一对文件句柄（描述符）。这对文件句柄指向一管道 i 节点。
    // 参数：filedes -文件句柄数组。fildes[0]用于读管道数据，fildes[1]向管道写入数据。
    // 成功时返回 0，出错时返回-1。
76  int sys_pipe(unsigned long * fildes)
77  {
78          struct m_inode * inode;
79          struct file * f[2];                     // 文件结构数组。
80          int fd[2];                              // 文件句柄数组。
81          int i,j;
82
    // 首先从系统文件表中取两个空闲项（引用计数字段为 0 的项），并分别设置引用计数为 1。
    // 若只有 1 个空闲项，则释放该项（引用计数复位）。若没有找到两个空闲项，则返回 -1。
83          j=0;
84          for(i=0;j<2 && i<NR_FILE;i++)
85                  if (!file_table[i].f_count)
86                          (f[j++]=i+file_table)->f_count++;
87          if (j==1)
88                  f[0]->f_count=0;
89          if (j<2)
90                  return -1;
    // 针对上面取得的两个文件表结构项，分别分配一文件句柄号，并使进程文件结构指针数组的
    // 两项分别指向这两个文件结构。而文件句柄即是该数组的索引号。类似地，如果只有一个空
    // 闲文件句柄，则释放该句柄（置空相应数组项）。如果没有找到两个空闲句柄，则释放上面
    // 获取的两个文件结构项（复位引用计数值），并返回 -1。
91          j=0;
92          for(i=0;j<2 && i<NR_OPEN;i++)
93                  if (!current->filp[i]) {
94                          current->filp[ fd[j]=i ] = f[j];
95                          j++;
96                  }
97          if (j==1)
98                  current->filp[fd[0]]=NULL;
99          if (j<2) {
100                 f[0]->f_count=f[1]->f_count=0;
101                 return -1;
102         }
    // 然后利用函数 get_pipe_inode()申请一个管道使用的 i 节点，并为管道分配一页内存作为缓
    // 冲区。如果不成功，则相应释放两个文件句柄和文件结构项，并返回-1。
103         if (!(inode=get_pipe_inode())) {          // fs/inode.c，第 231 行开始处。
104                 current->filp[fd[0]] =
105                         current->filp[fd[1]] = NULL;
106                 f[0]->f_count = f[1]->f_count = 0;
107                 return -1;
108         }
    // 如果管道 i 节点申请成功，则对两个文件结构进行初始化操作，让它们都指向同一个管道 i 节
    // 点，并把读写指针都置零。第 1 个文件结构的文件模式置为读，第 2 个文件结构的文件模式置
    // 为写。最后将文件句柄数组复制到对应的用户空间数组中，成功返回 0，退出。
109         f[0]->f_inode = f[1]->f_inode = inode;
110         f[0]->f_pos = f[1]->f_pos = 0;
```

```
111         f[0]->f_mode = 1;               /* read */
112         f[1]->f_mode = 2;               /* write */
113         put_fs_long(fd[0],0+fildes);
114         put_fs_long(fd[1],1+fildes);
115         return 0;
116 }
117
    //// 管道 io 控制函数。
    // 参数：pino - 管道 i 节点指针；cmd - 控制命令；arg - 参数。
    // 函数返回 0 表示执行成功，否则返回出错码。
118 int pipe_ioctl(struct m_inode *pino, int cmd, int arg)
119 {
    // 如果命令是取管道中当前可读数据长度，则把管道数据长度值添入用户参数指定的位置处，
    // 并返回 0。否则返回无效命令错误码。
120         switch (cmd) {
121                 case FIONREAD:
122                         verify_area((void *) arg,4);
123                         put_fs_long(PIPE_SIZE(*pino),(unsigned long *) arg);
124                         return 0;
125                 default:
126                         return -EINVAL;
127         }
128 }
129
```