```
/*
 * This file contains the procedures for the handling of select
 *
 * Created for Linux based loosely upon Mathius Lattner's minix
 * patches by Peter MacDonald. Heavily edited by Linus.
 */
/*
 * 本文件含有处理 select()系统调用的过程。
 *
 * 这是 Peter MacDonald 基于 Mathius Lattner 提供给 MINIX 系统的补丁
 * 程序修改而成。
 */

#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/tty.h>
#include <linux/sched.h>

#include <asm/segment.h>
#include <asm/system.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <const.h>
#include <errno.h>
#include <sys/time.h>
#include <signal.h>

/*
 * Ok, Peter made a complicated, but straightforward multiple_wait() function.
 * I have rewritten this, taking some shortcuts: This code may not be easy to
 * follow, but it should be free of race-conditions, and it's practical. If you
 * understand what I'm doing here, then you understand how the linux sleep/wakeup
 * mechanism works.
 *
 * Two very simple procedures, add_wait() and free_wait() make all the work. We
 * have to have interrupts disabled throughout the select, but that's not really
 * such a loss: sleeping automatically frees interrupts when we aren't in this
 * task.
 */
/*
 * OK，Peter 编制了复杂但很直观的多个_wait()函数。我对这些函数进行了改写，以使之
 * 更简洁：这些代码可能不容易看懂，但是其中应该不会存在竞争条件问题，并且很实际。
 * 如果你能理解这里编制的代码，那么就说明你已经理解 Linux 中睡眠/唤醒的工作机制。
 *
 * 两个很简单的过程，add_wait()和 free_wait()执行了主要操作。在整个 select 处理
 * 过程中我们不得不禁止中断。但是这样做并不会带来太多的损失：因为当我们不在执行
 * 本任务时睡眠状态会自动地释放中断（即其他任务会使用自己 EFLAGS 中的中断标志）。
 */

typedef struct {
```

```
38          struct task_struct * old_task;
39          struct task_struct ** wait_address;
40  } wait_entry;
41
42  typedef struct {
43          int nr;
44          wait_entry entry[NR_OPEN*3];
45  } select_table;
46
   // 把未准备好描述符的等待队列指针加入等待表 wait_table 中。参数*wait_address 是与描述
   // 符相关的等待队列头指针。例如 tty 读缓冲队列 secondary 的等待队列头指针是 proc_list。
   // 参数 p 是 do_select()中定义的等待表结构指针。
47  static void add_wait(struct task_struct ** wait_address, select_table * p)
48  {
49          int i;
50
   // 首先判断描述符是否有对应的等待队列，若无则返回。然后在等待表中搜索参数指定的等待
   // 队列指针是否已经在等待表中设置过，若设置过也立刻返回。这个判断主要是针对管道文件
   // 描述符。例如若一个管道在等待可以进行读操作，那么其必定可以立刻进行写操作。
51          if (!wait_address)
52                  return;
53          for (i = 0 ; i < p->nr ; i++)
54                  if (p->entry[i].wait_address == wait_address)
55                          return;
   // 然后我们把描述符对应等待队列的头指针保存在等待表 wait_table 中，同时让等待表项的
   // old_task 字段指向等待队列头指针指向的任务（若无则为 NULL），在让等待队列头指针指
   // 向当前任务。最后把等待表有效项计数值 nr 增 1（其在第 179 行被初始化为 0）。
56          p->entry[p->nr].wait_address = wait_address;
57          p->entry[p->nr].old_task = * wait_address;
58          *wait_address = current;
59          p->nr++;
60  }
61
   // 清空等待表。参数是等待表结构指针。本函数在 do_select()函数中睡眠后被唤醒返回时被调用
   // （第 204、207 行），用于唤醒等待表中处于各个等待队列上的其他任务。它与 kernel/sched.c
   // 中 sleep_on()函数的后半部分代码几乎完全相同，请参考对 sleep_on()函数的说明。
62  static void free_wait(select_table * p)
63  {
64          int i;
65          struct task_struct ** tpp;
66
   // 如果等待表中各项（供 nr 个有效项）记录的等待队列头指针表明还有其他后来添加进的等待
   // 任务（例如其他进程调用 sleep_on() 函数而睡眠在该等待队列上），则此时等待队列头指针
   // 指向的不是当前进程，那么我们就需要先唤醒这些任务。操作方法是将等待队列头所指任务先
   // 置为就绪状态（state = 0），并把自己设置为不可中断等待状态，即自己要等待这些后续进队
   // 列的任务被唤醒而执行时来唤醒本任务。然后重新执行调度程序。
67          for (i = 0; i < p->nr ; i++) {
68                  tpp = p->entry[i].wait_address;
69                  while (*tpp && *tpp != current) {
70                          (*tpp)->state = 0;
71                          current->state = TASK_UNINTERRUPTIBLE;
72                          schedule();
73                  }
```

```
    // 执行到这里，说明等待表当前处理项中的等待队列头指针字段 wait_address 指向当前任务，
    // 若它为空，则表明调度有问题，于是显示警告信息。然后我们让等待队列头指针指向在我们
    // 前面进入队列的任务（第 76 行）。若此时该头指针确实指向一个任务而不是 NULL，则说明
    // 队列中还有任务（*tpp 不为空），于是将该任务设置成就绪状态，唤醒之。最后把等待表的
    // 有效表项计数字段 nr 清零。
74                  if (!*tpp)
75                          printk("free_wait: NULL");
76                  if (*tpp = p->entry[i].old_task)
77                          (**tpp).state = 0;
78          }
79          p->nr = 0;
80  }
81
    // 根据文件 i 节点判断文件是否是字符终端设备文件。若是则返回其 tty 结构指针，否则返回 NULL。
82  static struct tty_struct * get_tty(struct m_inode * inode)
83  {
84          int major, minor;
85
    // 如果不是字符设备文件则返回 NULL。如果主设备号不是 5（控制终端）或 4，则返回 NULL。
86          if (!S_ISCHR(inode->i_mode))
87                  return NULL;
88          if ((major = MAJOR(inode->i_zone[0])) != 5 && major != 4)
89                  return NULL;
    // 如果主设备号是 5，那么其终端设备号等于进程的 tty 字段值，否则就等于字符设备文件次设备号。
    // 如果终端设备号小于 0，表示进程没有控制终端或没有使用终端，于是返回 NULL。否则返回对应的
    // tty 结构指针。
90          if (major == 5)
91                  minor = current->tty;
92          else
93                  minor = MINOR(inode->i_zone[0]);
94          if (minor < 0)
95                  return NULL;
96          return TTY_TABLE(minor);
97  }
98
99  /*
100  * The check_XX functions check out a file. We know it's either
101  * a pipe, a character device or a fifo (fifo's not implemented)
102  */
    /*
     * check_XX 函数用于检查一个文件。我们知道该文件要么是管道文件、
     * 要么是字符设备文件，或者要么是一个 FIFO（FIFO 还未实现）。
     */
    // 检查读文件操作是否准备好，即终端读缓冲队列 secondary 是否有字符可读，或者管道文件是否
    // 不空。参数 wait 是等待表指针；inode 是文件 i 节点指针。若描述符可进行读操作则返回 1，否
    // 则返回 0。
103 static int check_in(select_table * wait, struct m_inode * inode)
104 {
105         struct tty_struct * tty;
106
    // 首先根据文件 i 节点调用 get_tty()检测文件是否是一个 tty 终端（字符）设备文件，如果是则
    // 检查该终端读缓冲队列 secondary 中是否有字符可供读取，若有则返回 1，若此时 secondary 为
    // 空则把当前任务添加到 secondary 的等待队列 proc_list 上并返回 0。如果是管道文件，则判断
```

```
    // 目前管道中是否有字符可读，若有则返回 1，若没有（管道空）则把当前任务添加到管道 i 节点
    // 的等待队列上并返回 0。注意，PIPE_EMPTY()宏使用管道当前头尾指针位置来判断管道是否空。
    // 管道 i 节点的 i_zone[0]和 i_zone[1]字段分别存放着管道当前的头尾指针。
107         if (tty = get_tty(inode))
108                 if (!EMPTY(tty->secondary))
109                         return 1;
110                 else
111                         add_wait(&tty->secondary->proc_list, wait);
112         else if (inode->i_pipe)
113                 if (!PIPE_EMPTY(*inode))
114                         return 1;
115                 else
116                         add_wait(&inode->i_wait, wait);
117         return 0;
118 }
119
    // 检查文件写操作是否准备好，即终端写缓冲队列 write_q 中是否还有空闲位置可写，或者此时管
    // 道文件是否不满。参数 wait 是等待表指针；inode 是文件 i 节点指针。若描述符可进行写操作则
    // 返回 1，否则返回 0。
120 static int check_out(select_table * wait, struct m_inode * inode)
121 {
122         struct tty_struct * tty;
123
    // 首先根据文件 i 节点调用 get_tty()检测文件是否是一个 tty 终端（字符）设备文件，如果是则
    // 检查该终端写缓冲队列 write_q 中是否有空间可写入，若有则返回 1，若没有空空间则把当前任
    // 务添加到 write_q 的等待队列 proc_list 上并返回 0。如果是管道文件则判断目前管道中是否有
    // 空闲空间可写入字符，若有则返回 1，若没有（管道满）则把当前任务添加到管道 i 节点的等待
    // 队列上并返回 0。
124         if (tty = get_tty(inode))
125                 if (!FULL(tty->write_q))
126                         return 1;
127                 else
128                         add_wait(&tty->write_q->proc_list, wait);
129         else if (inode->i_pipe)
130                 if (!PIPE_FULL(*inode))
131                         return 1;
132                 else
133                         add_wait(&inode->i_wait, wait);
134         return 0;
135 }
136
    // 检查文件是否处于异常状态。对于终端设备文件，目前内核总是返回 0。对于管道文件，如果
    // 此时两个管道描述符中有一个或都已被关闭，则返回 1，否则就把当前任务添加到管道 i 节点
    // 的等待队列上并返回 0。返回 0。参数 wait 是等待表指针；inode 是文件 i 节点指针。若出现
    // 异常条件则返回 1，否则返回 0。
137 static int check_ex(select_table * wait, struct m_inode * inode)
138 {
139         struct tty_struct * tty;
140
141         if (tty = get_tty(inode))
142                 if (!FULL(tty->write_q))
143                         return 0;
144                 else
```

```
145                         return 0;
146         else if (inode->i_pipe)
147                 if (inode->i_count < 2)
148                         return 1;
149                 else
150                         add_wait(&inode->i_wait,wait);
151         return 0;
152 }
153
    // do_select() 是内核执行 select()系统调用的实际处理函数。该函数首先检查描述符集中各个
    // 描述符的有效性，然后分别调用相关描述符集描述符检查函数 check_XX()对每个描述符进行检
    // 查，同时统计描述符集中当前已经准备好的描述符个数。若有任何一个描述符已经准备好，本
    // 函数就会立刻返回，否则进程就会在本函数中进入睡眠状态，并在过了超时时间或者由于某个
    // 描述符所在等待队列上的进程被唤醒而使本进程继续运行。
154 int do_select(fd_set in, fd_set out, fd_set ex,
155         fd_set *inp, fd_set *outp, fd_set *exp)
156 {
157         int count;                                  // 已准备好的描述符个数计数值。
158         select_table wait_table;                    // 等待表结构。
159         int i;
160         fd_set mask;
161
    // 首先把 3 个描述符集进行或操作，在 mask 中得到描述符集中有效描述符比特位屏蔽码。 然后
    // 循环判断当前进程各个描述符是否有效并且包含在描述符集内。在循环中，每判断完一个描述
    // 符就会把 mask 右移 1 位，因此根据 mask 的最低有效比特位我们就可以判断相应描述符是否在
    // 用户给定的描述符集中。有效的描述符应该是一个管道文件描述符，或者是一个字符设备文件
    // 描述符，或者是一个 FIFO 描述符，其余类型的都作为无效描述符而返回 EBADF 错误。
162         mask = in | out | ex;
163         for (i = 0 ; i < NR_OPEN ; i++,mask >>= 1) {
164                 if (!(mask & 1))                    // 若不在描述符集中则继续判断下一个。
165                         continue;
166                 if (!current->filp[i])              // 若该文件未打开，则返回描述符错。
167                         return -EBADF;
168                 if (!current->filp[i]->f_inode)     // 若文件 i 节点指针空，则返回错误号。
169                         return -EBADF;
170                 if (current->filp[i]->f_inode->i_pipe) // 若是管道文件描述符，则有效。
171                         continue;
172                 if (S_ISCHR(current->filp[i]->f_inode->i_mode)) // 字符设备文件有效。
173                         continue;
174                 if (S_ISFIFO(current->filp[i]->f_inode->i_mode)) // FIFO 也有效。
175                         continue;
176                 return -EBADF;                      // 其余都作为无效描述符而返回。
177         }
    // 下面开始循环检查 3 个描述符集中的各个描述符是否准备好（可以操作）。此时 mask 用作当前
    // 正在处理描述符的屏蔽码。循环中的 3 个函数 check_in()、check_out()和 check_ex()分别用
    // 来判断描述符是否已经准备好。若一个描述符已经准备好，则在相关描述符集中设置对应比特
    // 位，并且把已准备好描述符个数计数值 count 增 1。第 183 行 for 循环语句中的 mask += mask
    // 等效于 mask<< 1。
178 repeat:
179         wait_table.nr = 0;
180         *inp = *outp = *exp = 0;
181         count = 0;
182         mask = 1;
```

```
183         for (i = 0 ; i < NR_OPEN ; i++, mask += mask) {
    // 如果此时判断的描述符在读操作描述符集中，并且该描述符已经准备好可以进行读操作，则把
    // 该描述符在描述符集 in 中对应比特位置为 1，同时把已准备好描述符个数计数值 count 增 1。
184             if (mask & in)
185                 if (check_in(&wait_table,current->filp[i]->f_inode)) {
186                     *inp |= mask;           // 描述符集中设置对应比特位。
187                     count++;                // 已准备好描述符个数计数。
188                 }
    // 如果此时判断的描述符在写操作描述符集中，并且该描述符已经准备好可以进行写操作，则把
    // 该描述符在描述符集 out 中对应比特位置为 1，同时把已准备好描述符个数计数值 count 增 1。
189             if (mask & out)
190                 if (check_out(&wait_table,current->filp[i]->f_inode)) {
191                     *outp |= mask;
192                     count++;
193                 }
    // 如果此时判断的描述符在异常描述符集中，并且该描述符已经有异常出现，则把该描述符在描
    // 述符集 ex 中对应比特位置为 1，同时把已准备好描述符个数计数值 count 增 1。
194             if (mask & ex)
195                 if (check_ex(&wait_table,current->filp[i]->f_inode)) {
196                     *exp |= mask;
197                     count++;
198                 }
199         }
    // 在对进程所有描述符判断处理过后，若没有发现有已准备好的描述符（count==0），并且此时
    // 进程没有收到任何非阻塞信号，并且此时有等待着的描述符或者等待时间还没有超时，那么我
    // 们就把当前进程状态设置成可中断睡眠状态，然后执行调度函数去执行其他任务。当内核又一
    // 次调度执行本任务时就调用 free_wait()唤醒相关等待队列上本任务前后的任务，然后跳转到
    // repeat 标号处（178 行）再次重新检测是否有我们关心的（描述符集中的）描述符已准备好。
200         if (!(current->signal & ~current->blocked) &&
201             (wait_table.nr || current->timeout) && !count) {
202                 current->state = TASK_INTERRUPTIBLE;
203                 schedule();
204                 free_wait(&wait_table);         // 本任务被唤醒返回后从这里开始执行。
205                 goto repeat;
206         }
    // 如果此时 count 不等于 0，或者接收到了信号，或者等待时间到并且没有需要等待的描述符，
    // 那么我们就调用 free_wait()唤醒等待队列上的任务，然后返回已准备好的描述符个数值。
207         free_wait(&wait_table);
208         return count;
209 }
210
211 /*
212  * Note that we cannot return -ERESTARTSYS, as we change our input
213  * parameters. Sad, but there you are. We could do some tweaking in
214  * the library function ...
215  */
    /*
     * 注意我们不能返回-ERESTARTSYS，因为我们会在 select 运行过程中改变
     * 输入参数值（*timeout）。很不幸，但你也只能接受这个事实。不过我们
     * 可以在库函数中做些处理...
     */
    // select 系统调用函数。该函数中的代码主要负责进行 select 功能操作前后的参数复制和转换
    // 工作。select 主要的工作由 do_select()函数来完成。sys_select()会首先根据参数传递来的
```

```
    // 缓冲区指针从用户数据空间把 select() 函数调用的参数分解复制到内核空间，然后设置需要
    // 等待的超时时间值 timeout，接着调用 do_select() 执行 select 功能，返回后就把处理结果
    // 再复制会用户空间中。
    // 参数 buffer 指向用户数据区中 select()函数的第 1 个参数处。如果返回值小于 0 表示执行时
    // 出现错误；如果返回值等于 0，则表示在规定等待时间内没有描述符准备好操作；如果返回值
    // 大于 0，则表示已准备好的描述符数量。
216 int sys_select( unsigned long *buffer )
217 {
218 /* Perform the select(nd, in, out, ex, tv) system call. */
    /* 执行 select(nd, in, out, ex, tv)系统调用 */
    // 首先定义几个局部变量，用于把指针参数传递来的 select()函数参数分解开来。
219         int i;
220         fd_set res_in, in = 0, *inp;                    // 读操作描述符集。
221         fd_set res_out, out = 0, *outp;                 // 写操作描述符集。
222         fd_set res_ex, ex = 0, *exp;                    // 异常条件描述符集。
223         fd_set mask;                                    // 处理的描述符数值范围（nd）屏蔽码。
224         struct timeval *tvp;                            // 等待时间结构指针。
225         unsigned long timeout;
226
    // 然后从用户数据区把参数分别隔离复制到局部指针变量中，并根据描述符集指针是否有效分别
    // 取得 3 个描述符集 in（读）、out（写）和 ex（异常）。其中 mask 也是一个描述符集变量，
    // 根据 3 个描述符集中最大描述符数值+1（即第 1 个参数 nd 的值），它被设置成用户程序关心的
    // 所有描述符的屏蔽码。例如，若 nd = 4，则 mask = 0b00001111（共 32 比特）。
227         mask = ~((~0) << get_fs_long(buffer++));
228         inp = (fd_set *) get_fs_long(buffer++);
229         outp = (fd_set *) get_fs_long(buffer++);
230         exp = (fd_set *) get_fs_long(buffer++);
231         tvp = (struct timeval *) get_fs_long(buffer);
232
233         if (inp)                                        // 若指针有效，则取读操作描述符集。
234                 in = mask & get_fs_long(inp);
235         if (outp)                                       // 若指针有效，则取写操作描述符集。
236                 out = mask & get_fs_long(outp);
237         if (exp)                                        // 若指针有效，则取异常描述符集。
238                 ex = mask & get_fs_long(exp);
    // 接下来我们尝试从时间结构中取出等待（睡眠）时间值 timeout。首先把 timeout 初始化成最大
    // （无限）值，然后从用户数据空间取得该时间结构中设置的时间值，经转换和加上系统当前滴答
    // 值 jiffies，最后得到需要等待的时间滴答数值 timeout。我们用此值来设置当前进程应该等待
    // 的延时。另外，第 241 行上 tv_usec 字段是微秒值，把它除以 1000000 后可得到对应秒数，再乘
    // 以系统每秒滴答数 HZ，即把 tv_usec 转换成滴答值。
239         timeout = 0xffffffff;
240         if (tvp) {
241                 timeout = get_fs_long((unsigned long *)&tvp->tv_usec)/(1000000/HZ);
242                 timeout += get_fs_long((unsigned long *)&tvp->tv_sec) * HZ;
243                 timeout += jiffies;
244         }
245         current->timeout = timeout;                     // 设置当前进程应该延时的滴答值。
    // select()函数的主要工作在 do_select()中完成。在调用该函数之后的代码用于把处理结果复制
    // 到用户数据区中，返回给用户。为了避免出现竞争条件，在调用 do_select()前需要禁止中断，
    // 并在该函数返回后再开启中断。
    // 如果在 do_select()返回之后进程的等待延时字段 timeout 还大于当前系统计时滴答值 jiffies，
    // 说明在超时之前已经有描述符准备好，于是这里我们先记下到超时还剩余的时间值，随后我们会
    // 把这个值返回给用户。如果进程的等待延时字段 timeout 已经小于或等于当前系统 jiffies，表
```

```
    // 示 do_select()可能是由于超时而返回，因此把剩余时间值设置为 0。
246         cli();                                  // 禁止响应中断。
247         i = do_select(in, out, ex, &res_in, &res_out, &res_ex);
248         if (current->timeout > jiffies)
249                 timeout = current->timeout - jiffies;
250         else
251                 timeout = 0;
252         sti();                                  // 开启中断响应。
    // 接下来我们把进程的超时字段清零。如果 do_select()返回的已准备好描述符个数小于 0，表示
    // 执行出错，于是返回这个错误号。然后我们把处理过的描述符集内容和延迟时间结构内容写回到
    // 用户数据缓冲空间。在写时间结构内容时还需要先将滴答时间单位表示的剩余延迟时间转换成秒
    // 和微秒值。
253         current->timeout = 0;
254         if (i < 0)
255                 return i;
256         if (inp) {
257                 verify_area(inp, 4);
258                 put_fs_long(res_in,inp);        // 可读描述符集。
259         }
260         if (outp) {
261                 verify_area(outp,4);
262                 put_fs_long(res_out,outp);      // 可写描述符集。
263         }
264         if (exp) {
265                 verify_area(exp,4);
266                 put_fs_long(res_ex,exp);        // 出现异常条件描述符集。
267         }
268         if (tvp) {
269                 verify_area(tvp, sizeof(*tvp));
270                 put_fs_long(timeout/HZ, (unsigned long *) &tvp->tv_sec);  // 秒。
271                 timeout %= HZ;
272                 timeout *= (1000000/HZ);
273                 put_fs_long(timeout, (unsigned long *) &tvp->tv_usec);  // 微秒。
274         }
    // 如果此时并没有已准备好的描述符，并且收到了某个非阻塞信号，则返回被中断错误号。
    // 否则返回已准备好的描述符个数值。
275         if (!i && (current->signal & ~current->blocked))
276                 return -EINTR;
277         return i;
278 }
279
```