程序 8-9 linux/kernel/sys.c 程序

```
 1 /*
 2  *  linux/kernel/sys.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
 9 #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。
10 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
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 <linux/config.h> // 内核常数配置文件。这里主要使用其中的系统名称常数符号信息。
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 <sys/times.h>    // 定义了进程中运行时间的结构 tms 以及 times()函数原型。
15 #include <sys/utsname.h>  // 系统名称结构头文件。
16 #include <sys/param.h>    // 系统参数头文件。含有系统一些全局常数符号。例如 HZ 等。
17 #include <sys/resource.h> // 系统资源头文件。含有有关进程资源使用情况的结构等信息。
18 #include <string.h>       // 字符串头文件。字符串或内存字节序列操作函数。
19
20 /*
21  * The timezone where the local system is located.  Used as a default by some
22  * programs who obtain this value by using gettimeofday.
23  */
   /*
    * 本系统所在的时区（timezone）。作为某些程序使用 gettimeofday 系统调用获取
    * 时区的默认值。
    */
   // 时区结构 timezone 第 1 个字段（tz_minuteswest）表示距格林尼治标准时间 GMT 以西的分钟
   // 数；第 2 个字段（tz_dsttime）是夏令时 DST（Daylight Savings Time）调整类型。 该结构
   // 定义在 include/sys/time.h 中。
24 struct timezone sys_tz = { 0, 0};
25
   // 根据进程组号 pgrp 取得进程组所属会话（session）号。该函数在 kernel/exit.c 中实现。
26 extern int session_of_pgrp(int pgrp);
27
   // 返回日期和时间（ftime - Fetch time）。
   // 以下返回值是-ENOSYS 的系统调用函数均表示在本版本内核中还未实现。
28 int sys_ftime()
29 {
30         return -ENOSYS;
31 }
32
33 int sys_break()
34 {
35         return -ENOSYS;
36 }
37
   // 用于当前进程对子进程进行调试(debugging)。
38 int sys_ptrace()
39 {
40         return -ENOSYS;
```

```
41 }
42
   // 改变并打印终端行设置。
43 int sys_stty()
44 {
45         return -ENOSYS;
46 }
47
   // 取终端行设置信息。
48 int sys_gtty()
49 {
50         return -ENOSYS;
51 }
52
   // 修改文件名。
53 int sys_rename()
54 {
55         return -ENOSYS;
56 }
57
58 int sys_prof()
59 {
60         return -ENOSYS;
61 }
62
63 /*
64  * This is done BSD-style, with no consideration of the saved gid, except
65  * that if you set the effective gid, it sets the saved gid too.  This
66  * makes it possible for a setgid program to completely drop its privileges,
67  * which is often a useful assertion to make when you are doing a security
68  * audit over a program.
69  *
70  * The general idea is that a program which uses just setregid() will be
71  * 100% compatible with BSD.  A program which uses just setgid() will be
72  * 100% compatible with POSIX w/ Saved ID's.
73  */
   /*
    * 以下是 BSD 形式的实现，没有考虑保存的 gid（saved gid 或 sgid），除了当你
    * 设置了有效的 gid（effective gid 或 egid）时，保存的 gid 也会被设置。这使
    * 得一个使用 setgid 的程序可以完全放弃其特权。当你在对一个程序进行安全审
    * 计时，这通常是一种很好的处理方法。
    *
    * 最基本的考虑是一个使用 setregid()的程序将会与 BSD 系统 100%的兼容。而一
    * 个使用 setgid()和保存的 gid 的程序将会与 POSIX 100%的兼容。
    */
   // 设置当前任务的实际以及/或者有效组 ID（gid）。如果任务没有超级用户特权，那么只能互
   // 换其实际组 ID 和有效组 ID。如果任务具有超级用户特权，就能任意设置有效的和实际的组
   // ID。保留的 gid（saved gid）被设置成与有效 gid。实际组 ID 是指进程当前的 gid。
74 int sys_setregid(int rgid, int egid)
75 {
76         if (rgid>0) {
77                 if ((current->gid == rgid) ||
78                     suser())
```

```
79                          current->gid = rgid;
80                  else
81                          return(-EPERM);
82          }
83          if (egid>0) {
84                  if ((current->gid == egid) ||
85                      (current->egid == egid) ||
86                      suser()) {
87                          current->egid = egid;
88                          current->sgid = egid;
89                  } else
90                          return(-EPERM);
91          }
92          return 0;
93  }
94
95  /*
96   * setgid() is implemeneted like SysV w/ SAVED_IDS
97   */
    /*
     * setgid()的实现与具有 SAVED_IDS 的 SYSV 的实现方法相似。
     */
    // 设置进程组号(gid)。如果任务没有超级用户特权，它可以使用 setgid() 将其有效 gid
    // （effective gid）设置为成其保留 gid(saved gid)或其实际 gid(real gid)。如果任务
    // 有超级用户特权，则实际 gid、有效 gid 和保留 gid 都被设置成参数指定的 gid。
98  int sys_setgid(int gid)
99  {
100         if (suser())
101                 current->gid = current->egid = current->sgid = gid;
102         else if ((gid == current->gid) || (gid == current->sgid))
103                 current->egid = gid;
104         else
105                 return -EPERM;
106         return 0;
107 }
108
    // 打开或关闭进程计帐功能。
109 int sys_acct()
110 {
111         return -ENOSYS;
112 }
113
    // 映射任意物理内存到进程的虚拟地址空间。
114 int sys_phys()
115 {
116         return -ENOSYS;
117 }
118
119 int sys_lock()
120 {
121         return -ENOSYS;
122 }
123
```

```
124 int sys_mpx()
125 {
126         return -ENOSYS;
127 }
128
129 int sys_ulimit()
130 {
131         return -ENOSYS;
132 }
133
    // 返回从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。如果 tloc 不为 null，
    // 则时间值也存储在那里。
    // 由于参数是一个指针，而其所指位置在用户空间，因此需要使用函数 put_fs_long() 来
    // 访问该值。在进入内核中运行时，段寄存器 fs 被默认地指向当前用户数据空间。因此该
    // 函数就可利用 fs 来访问用户空间中的值。
134 int sys_time(long * tloc)
135 {
136         int i;
137
138         i = CURRENT_TIME;
139         if (tloc) {
140                 verify_area(tloc,4);        // 验证内存容量是否够（这里是 4 字节）。
141                 put_fs_long(i,(unsigned long *)tloc);   // 放入用户数据段 tloc 处。
142         }
143         return i;
144 }
145
146 /*
147  * Unprivileged users may change the real user id to the effective uid
148  * or vice versa.  (BSD-style)
149  *
150  * When you set the effective uid, it sets the saved uid too.  This
151  * makes it possible for a setuid program to completely drop its privileges,
152  * which is often a useful assertion to make when you are doing a security
153  * audit over a program.
154  *
155  * The general idea is that a program which uses just setreuid() will be
156  * 100% compatible with BSD.  A program which uses just setuid() will be
157  * 100% compatible with POSIX w/ Saved ID's.
158  */
    /*
     * 无特权的用户可以见实际的 uid（real uid）改成有效的 uid（effective uid），
     * 反之也然。（BSD 形式的实现）
     *
     * 当你设置有效的 uid 时，它同时也设置了保存的 uid。这使得一个使用 setuid
     * 的程序可以完全放弃其特权。当你在对一个程序进行安全审计时，这通常是一种
     * 很好的处理方法。
     * 最基本的考虑是一个使用 setreuid()的程序将会与 BSD 系统 100%的兼容。而一
     * 个使用 setuid()和保存的 gid 的程序将会与 POSIX 100%的兼容。
     */
    // 设置任务的实际以及/或者有效的用户 ID（uid）。如果任务没有超级用户特权，那么只能
    // 互换其实际的 uid 和有效的 uid。如果任务具有超级用户特权，就能任意设置有效的和实
    // 际的用户 ID。保存的 uid（saved uid）被设置成与有效 uid 同值。
```

```
159 int sys_setreuid(int ruid, int euid)
160 {
161         int old_ruid = current->uid;
162
163         if (ruid>0) {
164                 if ((current->euid==ruid) ||
165                     (old_ruid == ruid) ||
166                     suser())
167                         current->uid = ruid;
168                 else
169                         return(-EPERM);
170         }
171         if (euid>0) {
172                 if ((old_ruid == euid) ||
173                     (current->euid == euid) ||
174                     suser()) {
175                         current->euid = euid;
176                         current->suid = euid;
177                 } else {
178                         current->uid = old_ruid;
179                         return(-EPERM);
180                 }
181         }
182         return 0;
183 }
184
185 /*
186  * setuid() is implemeneted like SysV w/ SAVED_IDS
187  *
188  * Note that SAVED_ID's is deficient in that a setuid root program
189  * like sendmail, for example, cannot set its uid to be a normal
190  * user and then switch back, because if you're root, setuid() sets
191  * the saved uid too.  If you don't like this, blame the bright people
192  * in the POSIX commmittee and/or USG.  Note that the BSD-style setreuid()
193  * will allow a root program to temporarily drop privileges and be able to
194  * regain them by swapping the real and effective uid.
195  */
    /*
     * setuid()的实现与具有 SAVED_IDS 的 SYSV 的实现方法相似。
     *
     * 请注意使用 SAVED_ID 的 setuid()在某些方面是不完善的。例如，一个使用
     * setuid 的超级用户程序 sendmail 就做不到把其 uid 设置成一个普通用户的
     * uid，然后再交换回来。 因为如果你是一个超级用户，setuid() 也同时会
     * 设置保存的 uid。如果你不喜欢这样的做法的话，就责怪 POSIX 组委会以及
     * /或者 USG 中的聪明人吧。不过请注意 BSD 形式的 setreuid()实现能够允许
     * 一个超级用户程序临时放弃特权，并且能通过交换实际的和有效的 uid 而
     * 再次获得特权。
     */
    // 设置任务用户 ID（uid）。如果任务没有超级用户特权，它可以使用 setuid()将其有效的
    // uid（effective uid）设置成其保存的 uid（saved uid）或其实际的 uid（real uid）。
    // 如果任务有超级用户特权，则实际的 uid、有效的 uid 和保存的 uid 都会被设置成参数指
    // 定的 uid。
196 int sys_setuid(int uid)
```

```
197 {
198         if (suser())
199                 current->uid = current->euid = current->suid = uid;
200         else if ((uid == current->uid) || (uid == current->suid))
201                 current->euid = uid;
202         else
203                 return -EPERM;
204         return(0);
205 }
206
    // 设置系统开机时间。参数 tptr 是从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。
    // 调用进程必须具有超级用户权限。其中 HZ=100，是内核系统运行频率。
    // 由于参数是一个指针，而其所指位置在用户空间，因此需要使用函数 get_fs_long()来访问该
    // 值。在进入内核中运行时，段寄存器 fs 被默认地指向当前用户数据空间。因此该函数就可利
    // 用 fs 来访问用户空间中的值。
    // 函数参数提供的当前时间值减去系统已经运行的时间秒值（jiffies/HZ）即是开机时间秒值。
207 int sys_stime(long * tptr)
208 {
209         if (!suser())                   // 如果不是超级用户则出错返回（许可）。
210                 return -EPERM;
211         startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;
212         jiffies_offset = 0;
213         return 0;
214 }
215
    // 获取当前任务运行时间统计值。
    // 在 tbuf 所指用户数据空间处返回 tms 结构的任务运行时间统计值。tms 结构中包括进程用户
    // 运行时间、内核（系统）时间、子进程用户运行时间、子进程系统运行时间。函数返回值是
    // 系统运行到当前的嘀嗒数。
216 int sys_times(struct tms * tbuf)
217 {
218         if (tbuf) {
219                 verify_area(tbuf,sizeof *tbuf);
220                 put_fs_long(current->utime,(unsigned long *)&tbuf->tms_utime);
221                 put_fs_long(current->stime,(unsigned long *)&tbuf->tms_stime);
222                 put_fs_long(current->cutime,(unsigned long *)&tbuf->tms_cutime);
223                 put_fs_long(current->cstime,(unsigned long *)&tbuf->tms_cstime);
224         }
225         return jiffies;
226 }
227
    // 当参数 end_data_seg 数值合理，并且系统确实有足够的内存，而且进程没有超越其最大数据
    // 段大小时，该函数设置数据段末尾为 end_data_seg 指定的值。该值必须大于代码结尾并且要
    // 小于堆栈结尾 16KB。返回值是数据段的新结尾值（如果返回值与要求值不同，则表明有错误
    // 发生）。该函数并不被用户直接调用，而由 libc 库函数进行包装，并且返回值也不一样。
228 int sys_brk(unsigned long end_data_seg)
229 {
    // 如果参数值大于代码结尾，并且小于（堆栈 - 16KB），则设置新数据段结尾值。
230         if (end_data_seg >= current->end_code &&
231             end_data_seg < current->start_stack - 16384)
232                 current->brk = end_data_seg;
233         return current->brk;            // 返回进程当前的数据段结尾值。
234 }
```

```
235
236 /*
237  * This needs some heave checking ...
238  * I just haven't get the stomach for it. I also don't fully
239  * understand sessions/pgrp etc. Let somebody who does explain it.
240  *
241  * OK, I think I have the protection semantics right.... this is really
242  * only important on a multi-user system anyway, to make sure one user
243  * can't send a signal to a process owned by another.  -TYT, 12/12/91
244  */
    /*
     * 下面代码需要某些严格的检查…
     * 我只是没有胃口来做这些。我也不完全明白 sessions/pgrp 等的含义。还是让
     * 了解它们的人来做吧。
     *
     * OK，我想我已经正确地实现了保护语义....。总之，这其实只对多用户系统是
     * 重要的，以确定一个用户不能向其他用户的进程发送信号。 -TYT 12/12/91
     */
    // 设置指定进程 pid 的进程组号为 pgid。
    // 参数 pid 是指定进程的进程号。如果它为 0，则让 pid 等于当前进程的进程号。参数 pgid
    // 是指定的进程组号。如果它为 0，则让它等于进程 pid 的进程组号。如果该函数用于将进程
    // 从一个进程组移到另一个进程组，则这两个进程组必须属于同一个会话(session)。在这种
    // 情况下，参数 pgid 指定了要加入的现有进程组 ID，此时该组的会话 ID 必须与将要加入进
    // 程的相同(263 行)。
245 int sys_setpgid(int pid, int pgid)
246 {
247         int i;
248
    // 如果参数 pid 为 0，则 pid 取值为当前进程的进程号 pid。如果参数 pgid 为 0，则 pgid 也
    // 取值为当前进程的 pid。[?? 这里与 POSIX 标准的描述有出入 ]。若 pgid 小于 0，则返回
    // 无效错误码。
249         if (!pid)
250                 pid = current->pid;
251         if (!pgid)
252                 pgid = current->pid;
253         if (pgid < 0)
254                 return -EINVAL;
    // 扫描任务数组，查找指定进程号 pid 的任务。如果找到了进程号是 pid 的进程，并且该进程
    // 的父进程就是当前进程或者该进程就是当前进程，那么若该任务已经是会话首领，则出错返回。
    // 若该任务的会话号（session）与当前进程的不同，或者指定的进程组号 pgid 与 pid 不同并且
    // pgid 进程组所属会话号与当前进程所属会话号不同，则也出错返回。 否则把查找到的进程的
    // pgrp 设置为 pgid，并返回 0。若没有找到指定 pid 的进程，则返回进程不存在出错码。
255         for (i=0 ; i<NR_TASKS ; i++)
256                 if (task[i] && (task[i]->pid == pid) &&
257                     ((task[i]->p_pptr == current) ||
258                      (task[i] == current))) {
259                    if (task[i]->leader)
260                            return -EPERM;
261                    if ((task[i]->session != current->session) ||
262                        ((pgid != pid) &&
263                         (session_of_pgrp(pgid) != current->session)))
264                            return -EPERM;
265                    task[i]->pgrp = pgid;
```

```
266                         return 0;
267                 }
268         return -ESRCH;
269 }
270
    // 返回当前进程的进程组号。与 getpgid(0)等同。
271 int sys_getpgrp(void)
272 {
273         return current->pgrp;
274 }
275
    // 创建一个会话(session)（即设置其 leader=1），并且设置其会话号=其组号=其进程号。
    // 如果当前进程已是会话首领并且不是超级用户，则出错返回。否则设置当前进程为新会话
    // 首领（leader = 1），并且设置当前进程会话号 session 和组号 pgrp 都等于进程号 pid，
    // 而且设置当前进程没有控制终端。最后系统调用返回会话号。
276 int sys_setsid(void)
277 {
278         if (current->leader && !suser())
279                 return -EPERM;
280         current->leader = 1;
281         current->session = current->pgrp = current->pid;
282         current->tty = -1;                      // 表示当前进程没有控制终端。
283         return current->pgrp;
284 }
285
286 /*
287  * Supplementary group ID's
288  */
    /*
     * 进程的其他用户组号。
     */
    // 取当前进程其他辅助用户组号。
    // 任务数据结构中 groups[]数组保存着进程同时所属的多个用户组号。该数组共 NGROUPS 个项，
    // 若某项的值是 NOGROUP（即为 -1），则表示从该项开始以后所有项都空闲。否则数组项中保
    // 存的是用户组号。
    // 参数 gidsetsize 是获取的用户组号个数；grouplist 是存储这些用户组号的用户空间缓存。
289 int sys_getgroups(int gidsetsize, gid_t *grouplist)
290 {
291         int     i;
292
    // 首先验证 grouplist 指针所指的用户缓存空间是否足够，然后从当前进程结构的 groups[]
    // 数组中逐个取得用户组号并复制到用户缓存中。在复制过程中，如果 groups[] 中的项数
    // 大于给定的参数 gidsetsize 所指定的个数，则表示用户给出的缓存太小，不能容下当前
    // 进程所有组号，因此此次取组号操作会出错返回。若复制过程正常，则函数最后会返回复
    // 制的用户组号个数。（gidsetsize - gid set size，用户组号集大小）。
293         if (gidsetsize)
294                 verify_area(grouplist, sizeof(gid_t) * gidsetsize);
295
296         for (i = 0; (i < NGROUPS) && (current->groups[i] != NOGROUP);
297              i++, grouplist++) {
298                 if (gidsetsize) {
299                         if (i >= gidsetsize)
300                                 return -EINVAL;
```

```
301                         put_fs_word(current->groups[i], (short *) grouplist);
302                 }
303         }
304         return(i);              // 返回实际含有的用户组号个数。
305 }
306
    // 设置当前进程同时所属的其他辅助用户组号。
    // 参数 gidsetsize 是将设置的用户组号个数；grouplist 是含有用户组号的用户空间缓存。
307 int sys_setgroups(int gidsetsize, gid_t *grouplist)
308 {
309         int     i;
310
    // 首先查权限和参数的有效性。只有超级用户可以修改或设置当前进程的辅助用户组号，而且
    // 设置的项数不能超过进程的 groups[NGROUPS]数组的容量。然后从用户缓冲中逐个复制用户
    // 组号，共 gidsetsize 个。如果复制的个数没有填满 groups[]，则在随后一项上填上值为-1
    // 的项（NOGROUP）。最后函数返回 0。
311         if (!suser())
312                 return -EPERM;
313         if (gidsetsize > NGROUPS)
314                 return -EINVAL;
315         for (i = 0; i < gidsetsize; i++, grouplist++) {
316                 current->groups[i] = get_fs_word((unsigned short *) grouplist);
317         }
318         if (i < NGROUPS)
319                 current->groups[i] = NOGROUP;
320         return 0;
321 }
322
    // 检查当前进程是否在指定的用户组 grp 中。是则返回 1，否则返回 0。
323 int in_group_p(gid_t grp)
324 {
325         int     i;
326
    // 如果当前进程的有效组号就是 grp，则表示进程属于 grp 进程组。函数返回 1。否则就在
    // 进程的辅助用户组数组中扫描是否有 grp 进程组号。若有则函数也返回 1。若扫描到值
    // 为 NOGROUP 的项，表示已扫描完全部有效项而没有发现匹配的组号，因此函数返回 0。
327         if (grp == current->egid)
328                 return 1;
329
330         for (i = 0; i < NGROUPS; i++) {
331                 if (current->groups[i] == NOGROUP)
332                         break;
333                 if (current->groups[i] == grp)
334                         return 1;
335         }
336         return 0;
337 }
338
    // utsname 结构含有一些字符串字段。用于保存系统的名称。其中包含 5 个字段，分别是：
    // 当前操作系统的名称、网络节点名称（主机名）、当前操作系统发行级别、操作系统版本
    // 号以及系统运行的硬件类型名称。该结构定义在 include/sys/utsname.h 文件中。 这里
    // 内核使用 include/linux/config.h 文件中的常数符号设置了它们的默认值。它们分别为
    // “Linux”，“(none)”，“0”，“0.12”，“i386”。
```

```
339 static struct utsname thisname = {
340         UTS_SYSNAME, UTS_NODENAME, UTS_RELEASE, UTS_VERSION, UTS_MACHINE
341 };
342
    // 获取系统名称等信息。
343 int sys_uname(struct utsname * name)
344 {
345         int i;
346
347         if (!name) return -ERROR;
348         verify_area(name,sizeof *name);
349         for(i=0;i<sizeof *name;i++)
350                 put_fs_byte(((char *) &thisname)[i],i+(char *) name);
351         return 0;
352 }
353
354 /*
355  * Only sethostname; gethostname can be implemented by calling uname()
356  */
    /*
     * 通过调用 uname()只能实现 sethostname 和 gethostname。
     */
    // 设置系统主机名（系统的网络节点名）。
    // 参数 name 指针指向用户数据区中含有主机名字符串的缓冲区；len 是主机名字符串长度。
357 int sys_sethostname(char *name, int len)
358 {
359         int     i;
360
    // 系统主机名只能由超级用户设置或修改，并且主机名长度不能超过最大长度 MAXHOSTNAMELEN。
361         if (!suser())
362                 return -EPERM;
363         if (len > MAXHOSTNAMELEN)
364                 return -EINVAL;
365         for (i=0; i < len; i++) {
366                 if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
367                         break;
368         }
    // 在复制完毕后，如果用户提供的字符串中没有包含 NULL 字符，那么若复制的主机名长度还没有
    // 超过 MAXHOSTNAMELEN，则在主机名字符串后添加一个 NULL。若已经填满 MAXHOSTNAMELEN 个字
    // 符，则把最后一个字符改成 NULL 字符。即 thisname.nodename[min(i,MAXHOSTNAMELEN)] = 0。
369         if (thisname.nodename[i]) {
370                 thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
371         }
372         return 0;
373 }
374
    // 取当前进程指定资源的界限值。
    // 进程的任务结构中定义有一个数组 rlim[RLIM_NLIMITS]，用于控制进程使用系统资源的界限。
    // 数组每个项是一个 rlimit 结构，其中包含两个字段。 一个说明进程对指定资源的当前限制
    // 界限（soft limit，即软限制），另一个说明系统对指定资源的最大限制界限（hard limit，
    // 即硬限制）。 rlim[] 数组的每一项对应系统对当前进程一种资源的界限信息。Linux 0.12
    // 系统共对 6 种资源规定了界限，即 RLIM_NLIMITS=6。请参考头文件 include/sys/resource.h
    // 中第 41 — 46 行的说明。
```

```
    // 参数 resource 指定我们咨询的资源名称，实际上它是任务结构中 rlim[]数组的索引项值。
    // 参数 rlim 是指向 rlimit 结构的用户缓冲区指针，用于存放取得的资源界限信息。
375 int sys_getrlimit(int resource, struct rlimit *rlim)
376 {
    // 所咨询的资源 resource 实际上是进程任务结构中 rlim[]数组的索引项值。该索引值当然不能
    // 大于数组的最大项数 RLIM_NLIMITS。在验证过 rlim 指针所指用户缓冲足够以后，这里就把
    // 参数指定的资源 resource 结构信息复制到用户缓冲中，并返回 0。
377         if (resource >= RLIM_NLIMITS)
378                 return -EINVAL;
379         verify_area(rlim,sizeof *rlim);
380         put_fs_long(current->rlim[resource].rlim_cur,       // 当前（软）限制值。
381                     (unsigned long *) rlim);
382         put_fs_long(current->rlim[resource].rlim_max,       // 系统（硬）限制值。
383                     ((unsigned long *) rlim)+1);
384         return 0;
385 }
386
    // 设置当前进程指定资源的界限值。
    // 参数 resource 指定我们设置界限的资源名称，实际上它是任务结构中 rlim[]数组的索引
    // 项值。参数 rlim 是指向 rlimit 结构的用户缓冲区指针，用于内核读取新的资源界限信息。
387 int sys_setrlimit(int resource, struct rlimit *rlim)
388 {
389         struct rlimit new, *old;
390
    // 首先判断参数 resource（任务结构 rlim[]项索引值）有效性。然后先让 rlimit 结构指针
    // old 指向指进程任务结构中指定资源的当前 rlimit 结构信息。接着把用户提供的资源界限
    // 信息复制到临时 rlimit 结构 new 中。此时如果判断出 new 结构中的软界限值或硬界限值
    // 大于进程该资源原硬界限值，并且当前不是超级用户的话，就返回许可错。否则表示 new
    // 中信息合理或者进程是超级用户进程，则修改原进程指定资源信息等于 new 结构中的信息，
    // 并成功返回 0。
391         if (resource >= RLIM_NLIMITS)
392                 return -EINVAL;
393         old = current->rlim + resource;         // 即 old = current->rlim[resource]。
394         new.rlim_cur = get_fs_long((unsigned long *) rlim);
395         new.rlim_max = get_fs_long(((unsigned long *) rlim)+1);
396         if (((new.rlim_cur > old->rlim_max) ||
397              (new.rlim_max > old->rlim_max)) &&
398             !suser())
399                 return -EPERM;
400         *old = new;
401         return 0;
402 }
403
404 /*
405  * It would make sense to put struct rusuage in the task_struct,
406  * except that would make the task_struct be *really big*.  After
407  * task_struct gets moved into malloc'ed memory, it would
408  * make sense to do this.  It will make moving the rest of the information
409  * a lot simpler!  (Which we're not doing right now because we're not
410  * measuring them yet).
411  */
    /*
     * 把 rusuage 结构放进任务结构 task_struct 中是恰当的，除非它会使任务
```

```
     * 结构长度变得非常大。在把任务结构移入内核 malloc 分配的内存中之后，
     * 这样做即使任务结构很大也没问题了。这将使得其余信息的移动变得非常
     * 方便！（我们还没有这样做，因为我们还没有测试过它们的大小）。
     */
    // 获取指定进程的资源利用信息。
    // 本系统调用提供当前进程或其已终止的和等待着的子进程资源使用情况。如果参数 who 等于
    // RUSAGE_SELF，则返回当前进程的资源利用信息。如果指定进程 who 是 RUSAGE_CHILDREN，
    // 则返回当前进程的已终止和等待着的子进程资源利用信息。 符号常数 RUSAGE_SELF 和
    // RUSAGE_CHILDREN 以及 rusage 结构都定义在 include/sys/resource.h 头文件中。
412 int sys_getrusage(int who, struct rusage *ru)
413 {
414         struct rusage r;
415         unsigned long   *lp, *lpend, *dest;
416
    // 首先判断参数指定进程的有效性。如果 who 既不是 RUSAGE_SELF（指定当前进程），也不是
    // RUSAGE_CHILDREN （指定子进程），则以无效参数码返回。否则在验证了指针 ru 指定的用
    // 户缓冲区域后，把临时 rusage 结构区域 r 清零。
417         if (who != RUSAGE_SELF && who != RUSAGE_CHILDREN)
418                 return -EINVAL;
419         verify_area(ru, sizeof *ru);
420         memset((char *) &r, 0, sizeof(r));      // 在 include/strings.h 文件最后。
    // 若参数 who 是 RUSAGE_SELF，则复制当前进程资源利用信息到 r 结构中。若指定进程 who
    // 是 RUSAGE_CHILDREN， 则复制当前进程的已终止和等待着的子进程资源利用信息到临时
    // rusuage 结构 r 中。宏 CT_TO_SECS 和 CT_TO_USECS 用于把系统当前嘀嗒数转换成用秒值
    // 加微秒值表示。它们定义在 include/linux/sched.h 文件中。 jiffies_offset 是系统
    // 嘀嗒数误差调整数。
421         if (who == RUSAGE_SELF) {
422                 r.ru_utime.tv_sec = CT_TO_SECS(current->utime);
423                 r.ru_utime.tv_usec = CT_TO_USECS(current->utime);
424                 r.ru_stime.tv_sec = CT_TO_SECS(current->stime);
425                 r.ru_stime.tv_usec = CT_TO_USECS(current->stime);
426         } else {
427                 r.ru_utime.tv_sec = CT_TO_SECS(current->cutime);
428                 r.ru_utime.tv_usec = CT_TO_USECS(current->cutime);
429                 r.ru_stime.tv_sec = CT_TO_SECS(current->cstime);
430                 r.ru_stime.tv_usec = CT_TO_USECS(current->cstime);
431         }
    // 然后让 lp 指针指向 r 结构，lpend 指向 r 结构末尾处，而 dest 指针指向用户空间中的 ru
    // 结构。最后把 r 中信息复制到用户空间 ru 结构中，并返回 0。
432         lp = (unsigned long *) &r;
433         lpend = (unsigned long *) (&r+1);
434         dest = (unsigned long *) ru;
435         for (; lp < lpend; lp++, dest++)
436                 put_fs_long(*lp, dest);
437         return(0);
438 }
439
    // 取得系统当前时间，并用指定格式返回。
    // timeval 结构和 timezone 结构都定义在 include/sys/time.h 文件中。timeval 结构含有秒
    // 和微秒（tv_sec 和 tv_usec）两个字段。timezone 结构含有本地距格林尼治标准时间以西
    // 的分钟数（tz_minuteswest）和夏令时间调整类型（tz_dsttime）两个字段。
    //  （dst -- Daylight Savings Time）
440 int sys_gettimeofday(struct timeval *tv, struct timezone *tz)
```

```
441 {
    // 如果参数给定的 timeval 结构指针不空，则在该结构中返回当前时间（秒值和微秒值）；
    // 如果参数给定的用户数据空间中 timezone 结构的指针不空，则也返回该结构的信息。
    // 程序中 startup_time 是系统开机时间（秒值）。 宏 CT_TO_SECS 和 CT_TO_USECS 用于
    // 把系统当前嘀嗒数转换成用秒值加微秒值表示。它们定义在 include/linux/sched.h
    // 文件中。jiffies_offset 是系统嘀嗒数误差调整数。
442         if (tv) {
443                 verify_area(tv, sizeof *tv);
444                 put_fs_long(startup_time + CT_TO_SECS(jiffies+jiffies_offset),
445                             (unsigned long *) tv);
446                 put_fs_long(CT_TO_USECS(jiffies+jiffies_offset),
447                             ((unsigned long *) tv)+1);
448         }
449         if (tz) {
450                 verify_area(tz, sizeof *tz);
451                 put_fs_long(sys_tz.tz_minuteswest, (unsigned long *) tz);
452                 put_fs_long(sys_tz.tz_dsttime, ((unsigned long *) tz)+1);
453         }
454         return 0;
455 }
456
457 /*
458  * The first time we set the timezone, we will warp the clock so that
459  * it is ticking GMT time instead of local time.  Presumably,
460  * if someone is setting the timezone then we are running in an
461  * environment where the programs understand about timezones.
462  * This should be done at boot time in the /etc/rc script, as
463  * soon as possible, so that the clock can be set right.  Otherwise,
464  * various programs will get confused when the clock gets warped.
465  */
    /*
     * 在第 1 次设置时区（timezone）时，我们会改变时钟值以让系统使用格林
     * 尼治标准时间（GMT）运行，而非使用本地时间。 推测起来说，如果某人
     * 设置了时区时间，那么我们就运行在程序知晓时区时间的环境中。设置时
     * 区操作应该在系统启动阶段，尽快地在/etc/rc 脚本程序中进行。这样时
     * 钟就可以设置正确。 否则的话，若我们以后才设置时区而导致时钟时间
     * 改变，可能会让一些程序的运行出现问题。
     */
    // 设置系统当前时间。
    // 参数 tv 是指向用户数据区中 timeval 结构信息的指针。参数 tz 是用户数据区中 timezone
    // 结构的指针。该操作需要超级用户权限。如果两者皆为空，则什么也不做，函数返回 0。
466 int sys_settimeofday(struct timeval *tv, struct timezone *tz)
467 {
468         static int      firsttime = 1;
469         void            adjust_clock();
470
    // 设置系统当前时间需要超级用户权限。如果 tz 指针不空，则设置系统时区信息。即复制用户
    // timezone 结构信息到系统中的 sys_tz 结构中（见第 24 行）。如果是第 1 次调用本系统调用
    // 并且参数 tv 指针不空，则调整系统时钟值。
471         if (!suser())
472                 return -EPERM;
473         if (tz) {
474                 sys_tz.tz_minuteswest = get_fs_long((unsigned long *) tz);
```

```
475                 sys_tz.tz_dsttime = get_fs_long(((unsigned long *) tz)+1);
476                 if (firsttime) {
477                         firsttime = 0;
478                         if (!tv)
479                                 adjust_clock();
480                 }
481         }
    // 如果参数的 timeval 结构指针 tv 不空，则用该结构信息设置系统时钟。首先从 tv 所指处
    // 获取以秒值（sec）加微秒值（usec）表示的系统时间，然后用秒值修改系统开机时间全局
    // 变量 startup_time 值，并用微秒值设置系统嘀嗒误差值 jiffies_offset。
482         if (tv) {
483                 int sec, usec;
484
485                 sec = get_fs_long((unsigned long *)tv);
486                 usec = get_fs_long(((unsigned long *)tv)+1);
487
488                 startup_time = sec - jiffies/HZ;
489                 jiffies_offset = usec * HZ / 1000000 - jiffies%HZ;
490         }
491         return 0;
492 }
493
494 /*
495  * Adjust the time obtained from the CMOS to be GMT time instead of
496  * local time.
497  *
498  * This is ugly, but preferable to the alternatives.  Otherwise we
499  * would either need to write a program to do it in /etc/rc (and risk
500  * confusion if the program gets run more than once; it would also be
501  * hard to make the program warp the clock precisely n hours)  or
502  * compile in the timezone information into the kernel.  Bad, bad....
503  *
504  * XXX Currently does not adjust for daylight savings time.  May not
505  * need to do anything, depending on how smart (dumb?) the BIOS
506  * is.  Blast it all.... the best thing to do not depend on the CMOS
507  * clock at all, but get the time via NTP or timed if you're on a
508  * network....                          - TYT, 1/1/92
509  */
    /*
     * 把从 CMOS 中读取的时间值调整为 GMT 时间值保存，而非本地时间值。
     *
     * 这里的做法很蹩脚，但要比其他方法好。否则我们就需要写一个程序并让它
     * 在/etc/rc 中运行来做这件事（并且冒着该程序可能会被多次执行而带来的
     * 问题。 而且这样做也很难让程序把时钟精确地调整 n 小时） 或者把时区信
     * 息编译进内核中。当然这样做就非常、非常差劲了...
     *
     * 目前下面函数（XXX）的调整操作并没有考虑到夏令时问题。依据 BIOS 有多
     * 么智能（愚蠢？）也许根本就不用考虑这方面。当然，最好的做法是完全不
     * 依赖于 CMOS 时钟，而是让系统通过 NTP（网络时钟协议）或者 timed（时间
     * 服务器）获得时间，如果机器联上网的话...。          - TYT，1/1/92
     */
    // 把系统启动时间调整为以 GMT 为标准的时间。
    // startup_time 是秒值，因此这里需要把时区分钟值乘上 60。
```

```
510 void adjust_clock()
511 {
512         startup_time += sys_tz.tz_minuteswest*60;
513 }
514
    // 设置当前进程创建文件属性屏蔽码为 mask & 0777。并返回原屏蔽码。
515 int sys_umask(int mask)
516 {
517         int old = current->umask;
518
519         current->umask = mask & 0777;
520         return (old);
521 }
522
```