```
/*
 *  linux/kernel/exit.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define DEBUG_PROC_TREE   // 定义符号“调试进程树”。

#include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
#include <signal.h>       // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
#include <sys/wait.h>     // 等待调用头文件。定义系统调用 wait()和 waitpid()及相关常数符号。

#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。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#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。
#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。

int sys_pause(void);      // 把进程置为睡眠状态，直到收到信号（kernel/sched.c，164 行）。
int sys_close(int fd);    // 关闭指定文件的系统调用（fs/open.c，219 行）。

//// 释放指定进程占用的任务槽及其任务数据结构占用的内存页面。
// 参数 p 是任务数据结构指针。该函数在后面的 sys_kill() 和 sys_waitpid() 函数中被调用。
// 扫描任务指针数组表 task[] 以寻找指定的任务。如果找到，则首先清空该任务槽，然后释放
// 该任务数据结构所占用的内存页面，最后执行调度函数并在返回时立即退出。如果在任务数组
// 表中没有找到指定任务对应的项，则内核 panic☺。
void release(struct task_struct * p)
{
        int i;

// 如果给定的任务结构指针为 NULL 则退出。如果该指针指向当前进程则显示警告信息退出。
        if (!p)
                return;
        if (p == current) {
                printk("task releasing itself\n\r");
                return;
        }
// 扫描任务结构指针数组，寻找指定的任务 p。如果找到，则置空任务指针数组中对应项，并且
// 更新任务结构之间的关联指针，释放任务 p 数据结构占用的内存页面。最后在执行调度程序
// 返回后退出。如果没有找到指定的任务 p，则说明内核代码出错了，则显示出错信息并死机。
// 更新链接部分的代码会把指定任务 p 从双向链表中删除。
        for (i=1 ; i<NR_TASKS ; i++)
                if (task[i]==p) {
                        task[i]=NULL;
                        /* Update links */    /* 更新链接 */
// 如果 p 不是最后（最老）的子进程，则让比其老的比邻进程指向比它新的比邻进程。如果 p
// 不是最新的子进程，则让比其新的比邻子进程指向比邻的老进程。 如果任务 p 就是最新的
// 子进程，则还需要更新其父进程的最新子进程指针 cptr 为指向 p 的比邻子进程。
// 指针 osptr（old sibling pointer）指向比 p 先创建的兄弟进程。
// 指针 ysptr（younger sibling pointer）指向比 p 后创建的兄弟进程。
// 指针 pptr（parent pointer）指向 p 的父进程。
// 指针 cptr（child pointer）是父进程指向最新（最后）创建的子进程。
                        if (p->p_osptr)
```

```
36                              p->p_osptr->p_ysptr = p->p_ysptr;
37                      if (p->p_ysptr)
38                              p->p_ysptr->p_osptr = p->p_osptr;
39                      else
40                              p->p_pptr->p_cptr = p->p_osptr;
41                      free_page((long)p);
42                      schedule();
43                      return;
44              }
45      panic("trying to release non-existent task");
46 }
47
48 #ifdef DEBUG_PROC_TREE
   // 如果定义了符号 DEBUG_PROC_TREE，则编译时包括以下代码。
49 /*
50  * Check to see if a task_struct pointer is present in the task[] array
51  * Return 0 if found, and 1 if not found.
52  */
   /*
    * 检查 task[]数组中是否存在一个指定的 task_struct 结构指针 p。
    * 如果存在则返回 0，否则返回 1。
    */
   // 检测任务结构指针 p。
53 int bad_task_ptr(struct task_struct *p)
54 {
55      int     i;
56
57      if (!p)
58              return 0;
59      for (i=0 ; i<NR_TASKS ; i++)
60              if (task[i] == p)
61                      return 0;
62      return 1;
63 }
64
65 /*
66  * This routine scans the pid tree and make sure the rep invarient still
67  * holds.  Used for debugging only, since it's very slow....
68  *
69  * It looks a lot scarier than it really is.... we're doing nothing more
70  * than verifying the doubly-linked list found in p_ysptr and p_osptr,
71  * and checking it corresponds with the process tree defined by p_cptr and
72  * p_pptr;
73  */
   /*
    * 下面的函数用于扫描进程树，以确定更改过的链接仍然正确。仅用于调式，
    * 因为该函数比较慢....
    *
    * 该函数看上去要比实际的恐怖.... 其实我们仅仅验证了指针 p_ysptr 和
    * p_osptr 构成的双向链表，并检查了链表与指针 p_cptr 和 p_pptr 构成的
    * 进程树之间的关系。
    */
   // 检查进程树。
```

```
74 void audit_ptree()
75 {
76         int     i;
77
   // 扫描系统中的除任务 0 以外的所有任务，检查它们中 4 个指针（pptr、cptr、ysptr 和 osptr）
   // 的正确性。若任务数组槽（项）为空则跳过。
78         for (i=1 ; i<NR_TASKS ; i++) {
79                 if (!task[i])
80                         continue;
   // 如果任务的父进程指针 p_pptr 没有指向任何进程（即在任务数组中不存在），则显示警告信息
   // “警告，pid 号 N 的父进程链接有问题”。以下语句对 cptr、ysptr 和 osptr 进行类似操作。
81                 if (bad_task_ptr(task[i]->p_pptr))
82                         printk("Warning, pid %d's parent link is bad\n",
83                                 task[i]->pid);
84                 if (bad_task_ptr(task[i]->p_cptr))
85                         printk("Warning, pid %d's child link is bad\n",
86                                 task[i]->pid);
87                 if (bad_task_ptr(task[i]->p_ysptr))
88                         printk("Warning, pid %d's ys link is bad\n",
89                                 task[i]->pid);
90                 if (bad_task_ptr(task[i]->p_osptr))
91                         printk("Warning, pid %d's os link is bad\n",
92                                 task[i]->pid);
   // 如果任务的父进程指针 p_pptr 指向了自己，则显示警告信息“警告，pid 号 N 的父进程链接
   // 指针指向了自己”。以下语句对 cptr、ysptr 和 osptr 进行类似操作。
93                 if (task[i]->p_pptr == task[i])
94                         printk("Warning, pid %d parent link points to self\n");
95                 if (task[i]->p_cptr == task[i])
96                         printk("Warning, pid %d child link points to self\n");
97                 if (task[i]->p_ysptr == task[i])
98                         printk("Warning, pid %d ys link points to self\n");
99                 if (task[i]->p_osptr == task[i])
100                        printk("Warning, pid %d os link points to self\n");
   // 如果任务有比自己先创建的比邻兄弟进程，那么就检查它们是否有共同的父进程，并检查这个
   // 老兄进程的 ysptr 指针是否正确地指向本进程。否则显示警告信息。
101                if (task[i]->p_osptr) {
102                        if (task[i]->p_pptr != task[i]->p_osptr->p_pptr)
103                                printk(
104                        "Warning, pid %d older sibling %d parent is %d\n",
105                                task[i]->pid, task[i]->p_osptr->pid,
106                                task[i]->p_osptr->p_pptr->pid);
107                        if (task[i]->p_osptr->p_ysptr != task[i])
108                                printk(
109                "Warning, pid %d older sibling %d has mismatched ys link\n",
110                                task[i]->pid, task[i]->p_osptr->pid);
111                }
   // 如果任务有比自己后创建的比邻兄弟进程，那么就检查它们是否有共同的父进程，并检查这个
   // 小弟进程的 osptr 指针是否正确地指向本进程。否则显示警告信息。
112                if (task[i]->p_ysptr) {
113                        if (task[i]->p_pptr != task[i]->p_ysptr->p_pptr)
114                                printk(
115                        "Warning, pid %d younger sibling %d parent is %d\n",
116                                task[i]->pid, task[i]->p_osptr->pid,
```

```
117                             task[i]->p_osptr->p_pptr->pid);
118                     if (task[i]->p_ysptr->p_osptr != task[i])
119                             printk(
120                 "Warning, pid %d younger sibling %d has mismatched os link\n",
121                             task[i]->pid, task[i]->p_ysptr->pid);
122                 }
    // 如果任务的最新子进程指针 cptr 不空，那么检查该子进程的父进程是否是本进程，并检查该
    // 子进程的小弟进程指针 yspter 是否为空。若不是则显示警告信息。
123                 if (task[i]->p_cptr) {
124                     if (task[i]->p_cptr->p_pptr != task[i])
125                             printk(
126                     "Warning, pid %d youngest child %d has mismatched parent link\n",
127                             task[i]->pid, task[i]->p_cptr->pid);
128                     if (task[i]->p_cptr->p_ysptr)
129                             printk(
130                     "Warning, pid %d youngest child %d has non-NULL ys link\n",
131                             task[i]->pid, task[i]->p_cptr->pid);
132                 }
133         }
134 }
135 #endif /* DEBUG_PROC_TREE */
136
    //// 向指定任务 p 发送信号 sig，权限为 priv。
    // 参数：sig - 信号值；p - 指定任务的指针；priv - 强制发送信号的标志。即不需要考虑进程
    // 用户属性或级别而能发送信号的权利。该函数首先判断参数的正确性，然后判断条件是否满足。
    // 如果满足就向指定进程发送信号 sig 并退出，否则返回未许可错误号。
137 static inline int send_sig(long sig,struct task_struct * p,int priv)
138 {
    // 如果没有权限，并且当前进程的有效用户 ID 与进程 p 的不同，并且也不是超级用户，则说明
    // 没有向 p 发送信号的权利。suser()定义为(current->euid==0)，用于判断是否是超级用户。
139         if (!p)
140                 return -EINVAL;
141         if (!priv && (current->euid!=p->euid) && !suser())
142                 return -EPERM;
    // 若需要发送的信号是 SIGKILL 或 SIGCONT，那么如果此时接收信号的进程 p 正处于停止状态
    // 就置其为就绪（运行）状态。然后修改进程 p 的信号位图 signal，去掉（复位）会导致进程
    // 停止的信号 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU。
143         if ((sig == SIGKILL) || (sig == SIGCONT)) {
144                 if (p->state == TASK_STOPPED)
145                         p->state = TASK_RUNNING;
146                 p->exit_code = 0;
147                 p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
148                                 (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
149         }
150         /* If the signal will be ignored, don't even post it */
        /* 如果要发送的信号 sig 将被进程 p 忽略掉，那么就根本不用发送 */
151         if ((int) p->sigaction[sig-1].sa_handler == 1)
152                 return 0;
153         /* Depends on order SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU */
        /* 以下判断依赖于 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 的次序 */
    // 如果信号是 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 之一，那么说明要让接收信号的进程 p
    // 停止运行。因此（若 p 的信号位图中有 SIGCONT 置位）就需要复位位图中继续运行的信号
    // SIGCONT 比特位。
```

```
154         if ((sig >= SIGSTOP) && (sig <= SIGTTOU))
155                 p->signal &= ~(1<<(SIGCONT-1));
156         /* Actually deliver the signal */
            /* 最后，我们向进程 p 发送信号 p */
157         p->signal |= (1<<(sig-1));
158         return 0;
159 }
160
    // 根据进程组号 pgrp 取得进程组所属的会话号。
    // 扫描任务数组，寻找进程组号为 pgrp 的进程，并返回其会话号。如果没有找到指定进程组号
    // 为 pgrp 的任何进程，则返回-1。
161 int session_of_pgrp(int pgrp)
162 {
163         struct task_struct **p;
164
165         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
166                 if ((*p)->pgrp == pgrp)
167                         return((*p)->session);
168         return -1;
169 }
170
    // 终止进程组（向进程组发送信号）。
    // 参数：pgrp - 指定的进程组号；sig - 指定的信号；priv - 权限。
    // 即向指定进程组 pgrp 中的每个进程发送指定信号 sig。只要向一个进程发送成功最后就会
    // 返回 0，否则如果没有找到指定进程组号 pgrp 的任何一个进程，则返回出错号-ESRCH，若
    // 找到进程组号是 pgrp 的进程，但是发送信号失败，则返回发送失败的错误码。
171 int kill_pg(int pgrp, int sig, int priv)
172 {
173         struct task_struct **p;
174         int err,retval = -ESRCH;          // -ESRCH 表示指定的进程不存在。
175         int found = 0;
176
    // 首先判断给定的信号和进程组号是否有效。然后扫描系统中所有任务。若扫描到进程组号为
    // pgrp 的进程，就向其发送信号 sig。只要有一次信号发送成功，函数最后就会返回 0。
177         if (sig<1 || sig>32 || pgrp<=0)
178                 return -EINVAL;
179         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
180                 if ((*p)->pgrp == pgrp) {
181                         if (sig && (err = send_sig(sig,*p,priv)))
182                                 retval = err;
183                         else
184                                 found++;
185                 }
186         return(found ? 0 : retval);
187 }
188
    // 终止进程（向进程发送信号）。
    // 参数：pid - 进程号；sig - 指定信号；priv - 权限。
    // 即向进程号为 pid 的进程发送指定信号 sig。若找到指定 pid 的进程，那么若信号发送成功，
    // 则返回 0，否则返回信号发送出错号。如果没有找到指定进程号 pid 的进程，则返回出错号
    // -ESRCH（指定进程不存在）。
189 int kill_proc(int pid, int sig, int priv)
190 {
```

```
191         struct task_struct **p;
192
193         if (sig<1 || sig>32)
194                 return -EINVAL;
195         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
196                 if ((*p)->pid == pid)
197                         return(sig ? send_sig(sig,*p,priv) : 0);
198         return(-ESRCH);
199 }
200
201 /*
202  * POSIX specifies that kill(-1,sig) is unspecified, but what we have
203  * is probably wrong.  Should make it like BSD or SYSV.
204  */
    /*
     * POSIX 标准指明 kill(-1,sig)未定义。但是我所知道的可能错了。应该让它
     * 象 BSD 或 SYSV 系统一样。
     */
    //// 系统调用 kill()可用于向任何进程或进程组发送任何信号，而并非只是杀死进程☺。
    // 参数 pid 是进程号；sig 是需要发送的信号。
    // 如果 pid 值>0，则信号被发送给进程号是 pid 的进程。
    // 如果 pid=0，那么信号就会被发送给当前进程的进程组中所有的进程。
    // 如果 pid=-1，则信号 sig 就会发送给除第一个进程（初始进程）外的所有进程。
    // 如果 pid < -1，则信号 sig 将发送给进程组-pid 的所有进程。
    // 如果信号 sig 为 0，则不发送信号，但仍会进行错误检查。如果成功则返回 0。
    // 该函数扫描任务数组表，并根据 pid 对满足条件的进程发送指定信号 sig。若 pid 等于 0，
    // 表明当前进程是进程组组长，因此需要向所有组内的进程强制发送信号 sig。
205 int sys_kill(int pid,int sig)
206 {
207         struct task_struct **p = NR_TASKS + task;    // p 指向任务数组最后一项。
208         int err, retval = 0;
209
210         if (!pid)
211                 return(kill_pg(current->pid,sig,0));
212         if (pid == -1) {
213                 while (--p > &FIRST_TASK)
214                         if (err = send_sig(sig,*p,0))
215                                 retval = err;
216                 return(retval);
217         }
218         if (pid < 0)
219                 return(kill_pg(-pid,sig,0));
220         /* Normal kill */
221         return(kill_proc(pid,sig,0));
222 }
223
224 /*
225  * Determine if a process group is "orphaned", according to the POSIX
226  * definition in 2.2.2.52.  Orphaned process groups are not to be affected
227  * by terminal-generated stop signals.  Newly orphaned process groups are
228  * to receive a SIGHUP and a SIGCONT.
229  *
230  * "I ask you, have you ever known what it is to be an orphan?"
```

```
231  */
     /*
      * 根据 POSIX 标准 2.2.2.52 节中的定义，确定一个进程组是否是“孤儿”。孤儿进程
      * 组不会受到终端产生的停止信号的影响。新近产生的孤儿进程组将会收到一个 SIGHUP
      * 信号和一个 SIGCONT 信号。
      *
      * “我问你，你是否真正知道作为一个孤儿意味着什么？”
      */
     // 以上提到的 POSIX P1003.1 2.2.2.52 节是关于孤儿进程组的描述。在两种情况下当一个进程
     // 终止时可能导致进程组变成“孤儿”。  一个进程组到其组外的父进程之间的联系依赖于该父
     // 进程和其子进程两者。因此，若组外最后一个连接父进程的进程或最后一个父进程的直接后裔
     // 终止的话，那么这个进程组就会成为一个孤儿进程组。在任何一种情况下，如果进程的终止导
     // 致进程组变成孤儿进程组，那么进程组中的所有进程就会与它们的作业控制 shell 断开联系。
     // 作业控制 shell 将不再具有该进程组存在的任何信息。而该进程组中处于停止状态的进程将会
     // 永远消失。为了解决这个问题，含有停止状态进程的新近产生的孤儿进程组就需要接收到一个
     // SIGHUP 信号和一个 SIGCONT 信号，用于指示它们已经从它们的会话（ session）中断开联系。
     // SIGHUP 信号将导致进程组中成员被终止，除非它们捕获或忽略了 SIGHUP 信号。而 SIGCONT 信
     // 号将使那些没有被 SIGHUP 信号终止的进程继续运行。 但在大多数情况下，如果组中有一个进
     // 程处于停止状态，那么组中所有的进程可能都处于停止状态。
     //
     // 判断一个进程组是否是孤儿进程。如果不是则返回 0；如果是则返回 1。
     // 扫描任务数组。如果任务项空，或者进程的组号与指定的不同，或者进程已经处于僵死状态，
     // 或者进程的父进程是 init 进程，则说明扫描的进程不是指定进程组的成员，或者不满足要求，
     // 于是跳过。 否则说明该进程是指定组的成员并且其父进程不是 init 进程。此时如果该进程
     // 父进程的组号不等于指定的组号 pgrp，但父进程的会话号等于进程的会话号，则说明它们同
     // 属于一个会话。因此指定的 pgrp 进程组肯定不是孤儿进程组。否则...。
232  int is_orphaned_pgrp(int pgrp)
233  {
234          struct task_struct **p;
235
236          for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
237                  if (!(*p) ||
238                      ((*p)->pgrp != pgrp) ||
239                      ((*p)->state == TASK_ZOMBIE) ||
240                      ((*p)->p_pptr->pid == 1))
241                          continue;
242                  if (((*p)->p_pptr->pgrp != pgrp) &&
243                      ((*p)->p_pptr->session == (*p)->session))
244                          return 0;
245          }
246          return(1);       /* (sighing) "Often!" */    /* （唉）是孤儿进程组！ */
247  }
248
     // 判断进程组中是否含有处于停止状态的作业（进程组）。有则返回 1；无则返回 0。
     // 查找方法是扫描整个任务数组。检查属于指定组 pgrp 的任何进程是否处于停止状态。
249  static int has_stopped_jobs(int pgrp)
250  {
251          struct task_struct ** p;
252
253          for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
254                  if ((*p)->pgrp != pgrp)
255                          continue;
256                  if ((*p)->state == TASK_STOPPED)
```

```
257                         return(1);
258         }
259         return(0);
260 }
261
    // 程序退出处理函数。在下面 365 行处被系统调用处理函数 sys_exit()调用。
    // 该函数将根据当前进程自身的特性对其进行处理，并把当前进程状态设置成僵死状态
    // TASK_ZOMBIE，最后调用调度函数 schedule()去执行其它进程，不再返回。
262 volatile void do_exit(long code)
263 {
264         struct task_struct *p;
265         int i;
266
    // 首先释放当前进程代码段和数据段所占的内存页。 函数 free_page_tables() 的第 1 个参数
    // （get_base()返回值）指明在 CPU 线性地址空间中起始基地址，第 2 个（get_limit()返回值）
    // 说明欲释放的字节长度值。get_base()宏中的 current->ldt[1]给出进程代码段描述符的位置
    // （current->ldt[2]给出进程代码段描述符的位置）；get_limit()中的 0x0f 是进程代码段的
    // 选择符（0x17 是进程数据段的选择符）。即在取段基地址时使用该段的描述符所处地址作为
    // 参数，取段长度时使用该段的选择符作为参数。 free_page_tables()函数位于 mm/memory.c
    // 文件的第 69 行开始处；get_base()和 get_limit()宏位于 include/linux/sched.h 头文件的第
    // 264 行开始处。
267         free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
268         free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    // 然后关闭当前进程打开着的所有文件。再对当前进程的工作目录 pwd、根目录 root、执行程序
    // 文件的 i 节点以及库文件进行同步操作，放回各个 i 节点并分别置空（释放）。 接着把当前
    // 进程的状态设置为僵死状态（TASK_ZOMBIE），并设置进程退出码。
269         for (i=0 ; i<NR_OPEN ; i++)
270                 if (current->filp[i])
271                         sys_close(i);
272         iput(current->pwd);
273         current->pwd = NULL;
274         iput(current->root);
275         current->root = NULL;
276         iput(current->executable);
277         current->executable = NULL;
278         iput(current->library);
279         current->library = NULL;
280         current->state = TASK_ZOMBIE;
281         current->exit_code = code;
282         /*
283          * Check to see if any process groups have become orphaned
284          * as a result of our exiting, and if they have any stopped
285          * jobs, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
286          *
287          * Case i: Our father is in a different pgrp than we are
288          * and we were the only connection outside, so our pgrp
289          * is about to become orphaned.
290          */
             /*
              * 检查当前进程的退出是否会造成任何进程组变成孤儿进程组。如果
              * 有，并且有处于停止状态（TASK_STOPPED）的组员，则向它们发送
              * 一个 SIGHUP 信号和一个 SIGCONT 信号。（POSIX 3.2.2.2 节要求）
              *
```

```
            * 情况 1：我们的父进程在另外一个与我们不同的进程组中，而本进程
            * 是我们与外界的唯一联系。所以我们的进程组将变成一个孤儿进程组。
            */
    // POSIX 3.2.2.2（1991 版）是关于 exit()函数的说明。如果父进程所在的进程组与当前进程的
    // 不同，但都处于同一个会话（session）中，并且当前进程所在进程组将要变成孤儿进程了并且
    // 当前进程的进程组中含有处于停止状态的作业（进程），那么就要向这个当前进程的进程组发
    // 送两个信号：SIGHUP 和 SIGCONT。发送这两个信号的原因见 232 行前的说明。
291         if ((current->p_pptr->pgrp != current->pgrp) &&
292             (current->p_pptr->session == current->session) &&
293             is_orphaned_pgrp(current->pgrp) &&
294             has_stopped_jobs(current->pgrp)) {
295                 kill_pg(current->pgrp,SIGHUP,1);
296                 kill_pg(current->pgrp,SIGCONT,1);
297         }
298         /* Let father know we died */         /* 通知父进程当前进程将终止 */
299         current->p_pptr->signal |= (1<<(SIGCHLD-1));
300
301         /*
302          * This loop does two things:
303          *
304          * A.   Make init inherit all the child processes
305          * B.   Check to see if any process groups have become orphaned
306          *      as a result of our exiting, and if they have any stopped
307          *      jons, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
308          */
            /*
             * 下面的循环做了两件事情：
             *
             * A.   让 init 进程继承当前进程所有子进程。
             * B.   检查当前进程的退出是否会造成任何进程组变成孤儿进程组。如果
             *      有，并且有处于停止状态（TASK_STOPPED）的组员，则向它们发送
             *      一个 SIGHUP 信号和一个 SIGCONT 信号。（POSIX 3.2.2.2 节要求）
             */
    // 如果当前进程有子进程（其 p_cptr 指针指向最近创建的子进程），则让进程 1（init 进程）
    // 成为其所有子进程的父进程。如果子进程已经处于僵死状态，则向 init 进程（父进程）发送
    // 子进程已终止信号 SIGCHLD。
309         if (p = current->p_cptr) {
310                 while (1) {
311                         p->p_pptr = task[1];
312                         if (p->state == TASK_ZOMBIE)
313                                 task[1]->signal |= (1<<(SIGCHLD-1));
314                         /*
315                          * process_group_orphan_check
316                          * Case ii: Our child is in a different pgrp
317                          * than we are, and it was the only connection
318                          * outside, so the child pgrp is now orphaned.
319                          */
                         /* 孤儿进程组检测。
                          * 情况 2：我们的子进程在不同的进程组中，而本进程
                          * 是它们唯一与外界的连接。因此现在子进程所在进程
                          * 组将变成孤儿进程组了。
                          */
    // 如果子进程与当前进程不在同一个进程组中但属于同一个 session 中，并且当前进程所在进程
```

```
    // 组将要变成孤儿进程了，并且当前进程的进程组中含有处于停止状态的作业（进程），那么就
    // 要向这个当前进程的进程组发送两个信号：SIGHUP 和 SIGCONT。  如果该子进程有兄弟进程，
    // 则继续循环处理这些兄弟进程。
320                     if ((p->pgrp != current->pgrp) &&
321                         (p->session == current->session) &&
322                         is_orphaned_pgrp(p->pgrp) &&
323                         has_stopped_jobs(p->pgrp)) {
324                             kill_pg(p->pgrp,SIGHUP,1);
325                             kill_pg(p->pgrp,SIGCONT,1);
326                     }
327                     if (p->p_osptr) {
328                             p = p->p_osptr;
329                             continue;
330                     }
331                     /*
332                      * This is it; link everything into init's children
333                      * and leave
334                      */
                     /*
                      * 就这样：将所有子进程链接成为 init 的子进程并退出循环。
                      */
    // 通过上面处理，当前进程子进程的所有兄弟子进程都已经处理过。此时 p 指向最老的兄弟子
    // 进程。于是把这些兄弟子进程全部加入 init 进程的子进程双向链表头部中。加入后，init
    // 进程的 p_cptr 指向当前进程原子进程中最年轻的（the youngest）子进程，而原子进程中
    // 最老的（the oldest）兄弟子进程 p_osptr 指向原 init 进程的最年轻进程，而原 init 进
    // 程中最年轻进程的 p_ysptr 指向原子进程中最老的兄弟子进程。最后把当前进程的 p_cptr
    // 指针置空，并退出循环。
335                     p->p_osptr = task[1]->p_cptr;
336                     task[1]->p_cptr->p_ysptr = p;
337                     task[1]->p_cptr = current->p_cptr;
338                     current->p_cptr = 0;
339                     break;
340             }
341     }
    // 如果当前进程是会话头领(leader)进程，那么若它有控制终端，则首先向使用该控制终端的
    // 进程组发送挂断信号 SIGHUP，然后释放该终端。接着扫描任务数组，把属于当前进程会话中
    // 进程的终端置空（取消）。
342     if (current->leader) {
343             struct task_struct **p;
344             struct tty_struct *tty;
345
346             if (current->tty >= 0) {
347                     tty = TTY_TABLE(current->tty);
348                     if (tty->pgrp>0)
349                             kill_pg(tty->pgrp, SIGHUP, 1);
350                     tty->pgrp = 0;
351                     tty->session = 0;
352             }
353             for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
354                     if ((*p)->session == current->session)
355                             (*p)->tty = -1;
356     }
    // 如果当前进程上次使用过协处理器，则把记录此信息的指针置空。若定义了调试进程树符号，
```

```
    // 则调用进程树检测显示函数。最后调用调度函数，重新调度进程运行，以让父进程能够处理
    // 僵死进程的其它善后事宜。
357         if (last_task_used_math == current)
358                 last_task_used_math = NULL;
359 #ifdef DEBUG_PROC_TREE
360         audit_ptree();
361 #endif
362         schedule();
363 }
364
    // 系统调用 exit()。终止进程。
    // 参数 error_code 是用户程序提供的退出状态信息，只有低字节有效。把 error_code 左移 8
    // 比特是 wait() 或 waitpid()函数的要求。低字节中将用来保存 wait()的状态信息。例如，
    // 如果进程处于暂停状态（TASK_STOPPED），那么其低字节就等于 0x7f。参见 sys/wait.h
    // 文件第 13--19 行。 wait() 或 waitpid() 利用这些宏就可以取得子进程的退出状态码或子
    // 进程终止的原因（信号）。
365 int sys_exit(int error_code)
366 {
367         do_exit((error_code&0xff)<<8);
368 }
369
    // 系统调用 waitpid()。挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止
    // 该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。如果 pid 所指的子进程早已
    // 退出（已成所谓的僵死进程），则本调用将立刻返回。子进程使用的所有资源将释放。
    // 如果 pid > 0，表示等待进程号等于 pid 的子进程。
    // 如果 pid = 0，表示等待进程组号等于当前进程组号的任何子进程。
    // 如果 pid < -1，表示等待进程组号等于 pid 绝对值的任何子进程。
    // 如果 pid = -1，表示等待任何子进程。
    // 若 options = WUNTRACED，表示如果子进程是停止的，也马上返回（无须跟踪）。
    // 若 options = WNOHANG，表示如果没有子进程退出或终止就马上返回。
    // 如果返回状态指针 stat_addr 不为空，则就将状态信息保存到那里。
    // 参数 pid 是进程号；*stat_addr 是保存状态信息位置的指针；options 是 waitpid 选项。
370 int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
371 {
372         int flag;               // 该标志用于后面表示所选出的子进程处于就绪或睡眠态。
373         struct task_struct *p;
374         unsigned long oldblocked;
375
    // 首先验证将要存放状态信息的位置处内存空间足够。然后复位标志 flag。接着从当前进程的最
    // 年轻子进程开始扫描子进程兄弟链表。
376         verify_area(stat_addr,4);
377 repeat:
378         flag=0;
379         for (p = current->p_cptr ; p ; p = p->p_osptr) {
    // 如果等待的子进程号 pid>0，但与被扫描子进程 p 的 pid 不相等，说明它是当前进程另外的子
    // 进程，于是跳过该进程，接着扫描下一个进程。
380                 if (pid>0) {
381                         if (p->pid != pid)
382                                 continue;
    // 否则，如果指定等待进程的 pid=0，表示正在等待进程组号等于当前进程组号的任何子进程。
    // 如果此时被扫描进程 p 的进程组号与当前进程的组号不等，则跳过。
383                 } else if (!pid) {
384                         if (p->pgrp != current->pgrp)
```

```
385                             continue;
    // 否则，如果指定的 pid < -1，表示正在等待进程组号等于 pid 绝对值的任何子进程。如果此时
    // 被扫描进程 p 的组号与 pid 的绝对值不等，则跳过。
386             } else if (pid != -1) {
387                     if (p->pgrp != -pid)
388                             continue;
389             }
    // 如果前 3 个对 pid 的判断都不符合，则表示当前进程正在等待其任何子进程，也即 pid = -1
    // 的情况。此时所选择到的进程 p 或者是其进程号等于指定 pid，或者是当前进程组中的任何
    // 子进程，或者是进程号等于指定 pid 绝对值的子进程，或者是任何子进程（此时指定的 pid
    // 等于 -1）。接下来根据这个子进程 p 所处的状态来处理。
    // 当子进程 p 处于停止状态时，如果此时参数选项 options 中 WUNTRACED 标志没有置位，表示
    // 程序无须立刻返回，或者子进程此时的退出码等于 0，于是继续扫描处理其他子进程。 如果
    // WUNTRACED 置位且子进程退出码不为 0，则把退出码移入高字节，或上状态信息 0x7f 后放入
    // *stat_addr，在复位子进程退出码后就立刻返回子进程号 pid。这里 0x7f 表示的返回状态使
    // WIFSTOPPED()宏为真。参见 include/sys/wait.h，14 行。
390             switch (p->state) {
391                     case TASK_STOPPED:
392                             if (!(options & WUNTRACED) ||
393                                 !p->exit_code)
394                                     continue;
395                             put_fs_long((p->exit_code << 8) | 0x7f,
396                                     stat_addr);
397                             p->exit_code = 0;
398                             return p->pid;
    // 如果子进程 p 处于僵死状态，则首先把它在用户态和内核态运行的时间分别累计到当前进程
    // （父进程）中，然后取出子进程的 pid 和退出码，把退出码放入返回状态位置 stat_addr 处
    // 并释放该子进程。最后返回子进程的退出码和 pid。 若定义了调试进程树符号，则调用进程
    // 树检测显示函数。
399                     case TASK_ZOMBIE:
400                             current->cutime += p->utime;
401                             current->cstime += p->stime;
402                             flag = p->pid;
403                             put_fs_long(p->exit_code, stat_addr);
404                             release(p);
405 #ifdef DEBUG_PROC_TREE
406                             audit_ptree();
407 #endif
408                             return flag;
    // 如果这个子进程 p 的状态既不是停止也不是僵死，那么就置 flag = 1。表示找到过一个符合
    // 要求的子进程，但是它处于运行态或睡眠态。
409                     default:
410                             flag=1;
411                             continue;
412             }
413         }
    // 在上面对任务数组扫描结束后，如果 flag 被置位，说明有符合等待要求的子进程并没有处
    // 于退出或僵死状态。此时如果已设置 WNOHANG 选项（表示若没有子进程处于退出或终止态就
    // 立刻返回），就立刻返回 0，退出。否则把当前进程置为可中断等待状态并，保留并修改
    // 当前进程信号阻塞位图，允许其接收到 SIGCHLD 信号。然后执行调度程序。当系统又开始
    // 执行本进程时，如果本进程收到除 SIGCHLD 以外的其他未屏蔽信号，则以退出码“重新启
    // 动系统调用”返回。否则跳转到函数开始处 repeat 标号处重复处理。
414         if (flag) {
```

```
415                 if (options & WNOHANG)
416                         return 0;
417                 current->state=TASK_INTERRUPTIBLE;
418                 oldblocked = current->blocked;
419                 current->blocked &= ~(1<<(SIGCHLD-1));
420                 schedule();
421                 current->blocked = oldblocked;
422                 if (current->signal & ~(current->blocked | (1<<(SIGCHLD-1))))
423                         return -ERESTARTSYS;
424                 else
425                         goto repeat;
426         }
    // 若 flag = 0，表示没有找到符合要求的子进程，则返回出错码（子进程不存在）。
427         return -ECHILD;
428 }
429
```