```
1  /*
2   *  linux/kernel/signal.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、初始任务 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。
8  #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
9  #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。
10
11 #include <signal.h>       // 信号头文件。定义信号符号常量，信号结构及信号操作函数原型。
12 #include <errno.h>        // 出错号头文件。定义出错号符号常量。
13
   // 获取当前任务信号屏蔽位图（屏蔽码或阻塞码）。sgetmask 可分解为 signal-get-mask。以下类似。
14 int sys_sgetmask()
15 {
16         return current->blocked;
17 }
18
   // 设置新的信号屏蔽位图。信号 SIGKILL 和 SIGSTOP 不能被屏蔽。返回值是原信号屏蔽位图。
19 int sys_ssetmask(int newmask)
20 {
21         int old=current->blocked;
22
23         current->blocked = newmask & ~(1<<(SIGKILL-1)) & ~(1<<(SIGSTOP-1));
24         return old;
25 }
26
   // 检测并取得进程收到的但被屏蔽（阻塞）的信号。还未处理信号的位图将被放入 set 中。
27 int sys_sigpending(sigset_t *set)
28 {
29     /* fill in "set" with signals pending but blocked. */
       /* 用还未处理并且被阻塞信号的位图填入 set 指针所指位置处 */
   // 首先验证进程提供的用户存储空间应有 4 个字节。然后把还未处理并且被阻塞信号的位图填入
   // set 指针所指位置处。
30     verify_area(set,4);
31     put_fs_long(current->blocked & current->signal, (unsigned long *)set);
32     return 0;
33 }
34
35 /* atomically swap in the new signal mask, and wait for a signal.
36  *
37  * we need to play some games with syscall restarting.  We get help
38  * from the syscall library interface.  Note that we need to coordinate
39  * the calling convention with the libc routine.
40  *
41  * "set" is just the sigmask as described in 1003.1-1988, 3.3.7.
42  *      It is assumed that sigset_t can be passed as a 32 bit quantity.
43  *
44  * "restart" holds a restart indication.  If it's non-zero, then we
45  *      install the old mask, and return normally.  If it's zero, we store
```

```
 *      the current mask in old_mask and block until a signal comes in.
 */
/* 自动地更换成新的信号屏蔽码，并等待信号的到来。
 *
 * 我们需要对系统调用（syscall）做一些处理。我们会从系统调用库接口取得某些信息。
 * 注意，我们需要把调用规则与 libc 库中的子程序统一考虑。
 *
 * "set" 正是 POSIX 标准 1003.1-1988 的 3.3.7 节中所描述的信号屏蔽码 sigmask。
 *       其中认为类型 sigset_t 能够作为一个 32 位量传递。
 *
 * "restart"中保持有重启指示标志。如果为非 0 值，那么我们就设置原来的屏蔽码，
 *       并且正常返回。如果它为 0，那么我们就把当前的屏蔽码保存在 oldmask 中
 *       并且阻塞进程，直到收到任何一个信号为止。
 */
// 该系统调用临时把进程信号屏蔽码替换成参数中给定的 set，然后挂起进程，直到收到一个
// 信号为止。
// restart 是一个被中断的系统调用重新启动标志。当第 1 次调用该系统调用时，它是 0。并且
// 在该函数中会把进程原来的阻塞码 blocked 保存起来（old_mask），并设置 restart 为非 0
// 值。因此当进程第 2 次调用该系统调用时，它就会恢复进程原来保存在 old_mask 中的阻塞码。
int sys_sigsuspend(int restart, unsigned long old_mask, unsigned long set)
{
// pause()系统调用将导致调用它的进程进入睡眠状态，直到收到一个信号。该信号或者会终止
// 进程的执行，或者导致进程去执行相应的信号捕获函数。
    extern int sys_pause(void);

// 如果 restart 标志不为 0，表示是重新让程序运行起来。于是恢复前面保存在 old_mask 中的
// 原进程阻塞码。并返回码-EINTR（系统调用被信号中断）。
    if (restart) {
        /* we're restarting */      /* 我们正在重新启动系统调用 */
        current->blocked = old_mask;
        return -EINTR;
    }
// 否则表示 restart 标志的值是 0。表示第 1 次调用。于是首先设置 restart 标志（置为 1），
// 保存进程当前阻塞码 blocked 到 old_mask 中，并把进程的阻塞码替换成 set。然后调用
// pause()让进程睡眠，等待信号的到来。当进程收到一个信号时，pause() 就会返回，并且
// 进程会去执行信号处理函数，然后本调用返回 -ERESTARTNOINTR 码退出。这个返回码说明
// 在处理完信号后要求返回到本系统调用中继续运行，即本系统调用不会被中断。
    /* we're not restarting.  do the work */
    /* 我们不是重新重新运行，那么就干活吧 */
    *(&restart) = 1;
    *(&old_mask) = current->blocked;
    current->blocked = set;
    (void) sys_pause();                 /* return after a signal arrives */
    return -ERESTARTNOINTR;             /* handle the signal, and come back */
}

// 复制 sigaction 数据到 fs 数据段 to 处。即从内核空间复制到用户（任务）数据段中。
static inline void save_old(char * from,char * to)
{
        int i;

// 首先验证 to 处的内存空间是否足够大。然后把一个 sigaction 结构信息复制到 fs 段（用户）
// 空间中。宏函数 put_fs_byte()在 include/asm/segment.h 中实现。
```

```
69          verify_area(to, sizeof(struct sigaction));
70          for (i=0 ; i< sizeof(struct sigaction) ; i++) {
71                  put_fs_byte(*from,to);
72                  from++;
73                  to++;
74          }
75  }
76
    // 把 sigaction 数据从 fs 数据段 from 位置复制到 to 处。即从用户数据空间取到内核数据段中。
77  static inline void get_new(char * from,char * to)
78  {
79          int i;
80
81          for (i=0 ; i< sizeof(struct sigaction) ; i++)
82                  *(to++) = get_fs_byte(from++);
83  }
84
    // signal()系统调用。类似于 sigaction()。为指定的信号安装新的信号句柄(信号处理程序)。
    // 信号句柄可以是用户指定的函数，也可以是 SIG_DFL（默认句柄）或 SIG_IGN（忽略）。
    // 参数 signum --指定的信号；handler -- 指定的句柄；restorer -恢复函数指针，该函数由
    // Libc 库提供。用于在信号处理程序结束后恢复系统调用返回时几个寄存器的原有值以及系统
    // 调用的返回值，就好象系统调用没有执行过信号处理程序而直接返回到用户程序一样。 函数
    // 返回原信号句柄。
85  int sys_signal(int signum, long handler, long restorer)
86  {
87          struct sigaction tmp;
88
    // 首先验证信号值在有效范围（1--32）内，并且不得是信号 SIGKILL（和 SIGSTOP）。因为这
    // 两个信号不能被进程捕获。
89          if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
90                  return -EINVAL;
    // 然后根据提供的参数组建 sigaction 结构内容。sa_handler 是指定的信号处理句柄（函数）。
    // sa_mask 是执行信号处理句柄时的信号屏蔽码。sa_flags 是执行时的一些标志组合。这里设定
    // 该信号处理句柄只使用 1 次后就恢复到默认值，并允许信号在自己的处理句柄中收到。
91          tmp.sa_handler = (void (*)(int)) handler;
92          tmp.sa_mask = 0;
93          tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
94          tmp.sa_restorer = (void (*)(void)) restorer;        // 保存恢复处理函数指针。
    // 接着取该信号原来的处理句柄，并设置该信号的 sigaction 结构。最后返回原信号句柄。
95          handler = (long) current->sigaction[signum-1].sa_handler;
96          current->sigaction[signum-1] = tmp;
97          return handler;
98  }
99
    // sigaction()系统调用。改变进程在收到一个信号时的操作。signum 是除了 SIGKILL 以外的
    // 任何信号。[如果新操作（action）不为空 ]则新操作被安装。如果 oldaction 指针不为空，
    // 则原操作被保留到 oldaction。成功则返回 0，否则为-EINVAL。
100 int sys_sigaction(int signum, const struct sigaction * action,
101         struct sigaction * oldaction)
102 {
103         struct sigaction tmp;
104
    // 首先验证信号值在有效范围（1--32）内，并且不得是信号 SIGKILL（和 SIGSTOP）。因为这
```

```
    // 两个信号不能被进程捕获。
105         if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
106                 return -EINVAL;
    // 在信号的 sigaction 结构中设置新的操作（动作）。如果 oldaction 指针不为空的话，则将
    // 原操作指针保存到 oldaction 所指的位置。
107         tmp = current->sigaction[signum-1];
108         get_new((char *) action,
109                 (char *) (signum-1+current->sigaction));
110         if (oldaction)
111                 save_old((char *) &tmp,(char *) oldaction);
    // 如果允许信号在自己的信号句柄中收到，则令屏蔽码为 0，否则设置屏蔽本信号。
112         if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
113                 current->sigaction[signum-1].sa_mask = 0;
114         else
115                 current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
116         return 0;
117 }
118
119 /*
120  * Routine writes a core dump image in the current directory.
121  * Currently not implemented.
122  */
    /*
     * 在当前目录中产生 core dump 映像文件的子程序。目前还没有实现。
     */
123 int core_dump(long signr)
124 {
125         return(0);      /* We didn't do a dump */
126 }
127
    // 系统调用的中断处理程序中真正的信号预处理程序（在 kernel/sys_call.s,119 行）。这段
    // 代码的主要作用是将信号处理句柄插入到用户程序堆栈中，并在本系统调用结束返回后立刻
    // 执行信号句柄程序，然后继续执行用户的程序。
    // 函数的参数是进入系统调用处理程序 sys_call.s 开始，直到调用本函数（sys_call.s
    // 第 125 行）前逐步压入堆栈的值。这些值包括（在 sys_call.s 中的代码行）：
    // ① CPU 执行中断指令压入的用户栈地址 ss 和 esp、标志寄存器 eflags 和返回地址 cs 和 eip；
    // ② 第 85--91 行在刚进入 system_call 时压入栈的段寄存器 ds、es、fs 以及寄存器 eax
    //  （orig_eax）、edx、ecx 和 ebx 的值；
    // ③ 第 100 行调用 sys_call_table 后压入栈中的相应系统调用处理函数的返回值（eax）。
    // ④ 第 124 行压入栈中的当前处理的信号值（signr）。
128 int do_signal(long signr,long eax,long ebx, long ecx, long edx, long orig_eax,
129         long fs, long es, long ds,
130         long eip, long cs, long eflags,
131         unsigned long * esp, long ss)
132 {
133         unsigned long sa_handler;
134         long old_eip=eip;
135         struct sigaction * sa = current->sigaction + signr - 1;
136         int longs;                                  // 即 current->sigaction[signr-1]。
137
138         unsigned long * tmp_esp;
139
    // 以下是调试语句。当定义了 notdef 时会打印相关信息。
```

```
140 #ifdef notdef
141         printk("pid: %d, signr: %x, eax=%d, oeax = %d, int=%d\n",
142                 current->pid, signr, eax, orig_eax,
143                 sa->sa_flags & SA_INTERRUPT);
144 #endif
    // 如果不是系统调用而是其它中断执行过程中调用到本函数时， roig_eax 值为 -1。 参见
    // sys_call.s 第 144 行 等语句。因此当 orig_eax 不等于 -1 时，说明是在某个系统调用的
    // 最后调用了本函数。在 kernel/exit.c 的 waitpid() 函数中，如果收到了 SIGCHLD 信号，
    // 或者在读管道函数 fs/pipe.c 中管道当前读数据但没有读到任何数据等情况下，进程收到
    // 了任何一个非阻塞的信号，则都会以 -ERESTARTSYS 返回值返回。它表示进程可以被中断，
    // 但是在继续执行后会重新启动系统调用。返回码-ERESTARTNOINTR 说明在处理完信号后要求
    // 返回到原系统调用中继续运行，即系统调用不会被中断。参见前面第 62 行。
    // 因此下面语句说明如果是在系统调用中调用的本函数，并且相应系统调用的返回码 eax 等于
    // -ERESTARTSYS 或 -ERESTARTNOINTR 时进行下面的处理（实际上还没有真正回到用户程序中）。
145         if ((orig_eax != -1) &&
146             ((eax == -ERESTARTSYS) || (eax == -ERESTARTNOINTR))) {
    // 如果系统调用返回码是 -ERESTARTSYS（重新启动系统调用），并且 sigaction 中含有标志
    // SA_INTERRUPT（系统调用被信号中断后不重新启动系统调用）或者信号值小于 SIGCONT 或者
    // 信号值大于 SIGTTOU（即信号不是 SIGCONT、SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU），则
    // 修改系统调用的返回值为 eax = -EINTR，即被信号中断的系统调用。
147                 if ((eax == -ERESTARTSYS) && ((sa->sa_flags & SA_INTERRUPT) ||
148                     signr < SIGCONT || signr > SIGTTOU))
149                         *(&eax) = -EINTR;
150                 else {
    // 否则就恢复进程寄存器 eax 在调用系统调用之前的值，并且把原程序指令指针回调 2 字节。即
    // 当返回用户程序时，让程序重新启动执行被信号中断的系统调用。
151                         *(&eax) = orig_eax;
152                         *(&eip) = old_eip -= 2;
153                 }
154         }
    // 如果信号句柄为 SIG_IGN（1，默认忽略句柄）则不对信号进行处理而直接返回。
155         sa_handler = (unsigned long) sa->sa_handler;
156         if (sa_handler==1)
157                 return(1);   /* Ignore, see if there are more signals... */
    // 如果句柄为 SIG_DFL（0，默认处理），则根据具体的信号进行分别处理。
158         if (!sa_handler) {
159                 switch (signr) {
    // 如果信号是以下两个则也忽略之，并返回。
160                 case SIGCONT:
161                 case SIGCHLD:
162                         return(1);  /* Ignore, ... */
163
    // 如果信号是以下 4 种信号之一，则把当前进程状态置为停止状态 TASK_STOPPED。若当前进程
    // 父进程对 SIGCHLD 信号的 sigaction 处理标志 SA_NOCLDSTOP （即当子进程停止执行或又继
    // 续执行时不要产生 SIGCHLD 信号）没有置位，那么就给父进程发送 SIGCHLD 信号。
164                 case SIGSTOP:
165                 case SIGTSTP:
166                 case SIGTTIN:
167                 case SIGTTOU:
168                         current->state = TASK_STOPPED;
169                         current->exit_code = signr;
170                         if (!(current->p_pptr->sigaction[SIGCHLD-1].sa_flags &
171                                         SA_NOCLDSTOP))
```

```
172                             current->p_pptr->signal |= (1<<(SIGCHLD-1));
173                     return(1);  /* Reschedule another event */
174
    // 如果信号是以下 6 种信号之一，那么若信号产生了 core dump，则以退出码为 signr|0x80
    // 调用 do_exit()退出。否则退出码就是信号值。do_exit()的参数是返回码和程序提供的退出
    // 状态信息。可作为 wait()或 waitpid()函数的状态信息。参见 sys/wait.h 文件第 13-18 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）。
175                 case SIGQUIT:
176                 case SIGILL:
177                 case SIGTRAP:
178                 case SIGIOT:
179                 case SIGFPE:
180                 case SIGSEGV:
181                         if (core_dump(signr))
182                                 do_exit(signr|0x80);
183                         /* fall through */
184                 default:
185                         do_exit(signr);
186                 }
187         }
188         /*
189          * OK, we're invoking a handler
190          */
            /*
             * OK，现在我们准备对信号句柄调用的设置
             */
    // 如果该信号句柄只需被调用一次，则将该句柄置空。注意，该信号句柄在前面已经保存在
    // sa_handler 指针中。
    // 在系统调用进入内核时，用户程序返回地址（eip、cs）被保存在内核态栈中。下面这段代
    // 码修改内核态堆栈上用户调用系统调用时的代码指针 eip 为指向信号处理句柄，同时也将
    // sa_restorer、signr、进程屏蔽码(如果 SA_NOMASK 没置位)、eax、ecx、edx 作为参数以及
    // 原调用系统调用的程序返回指针及标志寄存器值压入用户堆栈。 因此在本次系统调用中断
    // 返回用户程序时会首先执行用户的信号句柄程序，然后再继续执行用户程序。
191         if (sa->sa_flags & SA_ONESHOT)
192                 sa->sa_handler = NULL;
    // 将内核态栈上用户调用系统调用下一条代码指令指针 eip 指向该信号处理句柄。由于 C 函数
    // 是传值函数，因此给 eip 赋值时需要使用 "*(&eip)" 的形式。另外，如果允许信号自己的
    // 处理句柄收到信号自己，则也需要将进程的阻塞码压入堆栈。
    // 这里请注意，使用如下方式（第 193 行）对普通 C 函数参数进行修改是不起作用的。因为当
    // 函数返回时堆栈上的参数将会被调用者丢弃。这里之所以可以使用这种方式，是因为该函数
    // 是从汇编程序中被调用的，并且在函数返回后汇编程序并没有把调用 do_signal()时的所有
    // 参数都丢弃。eip 等仍然在堆栈中。
    // sigaction 结构的 sa_mask 字段给出了在当前信号句柄（信号描述符）程序执行期间应该被
    // 屏蔽的信号集。同时，引起本信号句柄执行的信号也会被屏蔽。 不过若 sa_flags 中使用了
    // SA_NOMASK 标志，那么引起本信号句柄执行的信号将不会被屏蔽掉。如果允许信号自己的处
    // 理句柄程序收到信号自己，则也需要将进程的信号阻塞码压入堆栈。
193         *(&eip) = sa_handler;
194         longs = (sa->sa_flags & SA_NOMASK)?7:8;
    // 将原调用程序的用户堆栈指针向下扩展 7（或 8）个长字（用来存放调用信号句柄的参数等），
    // 并检查内存使用情况（例如如果内存超界则分配新页等）。
195         *(&esp) -= longs;
196         verify_area(esp,longs*4);
    // 在用户堆栈中从下到上存放 sa_restorer、信号 signr、屏蔽码 blocked（如果 SA_NOMASK
```

```
    // 置位）、eax、ecx、edx、eflags 和用户程序原代码指针。
197         tmp_esp=esp;
198         put_fs_long((long) sa->sa_restorer,tmp_esp++);
199         put_fs_long(signr,tmp_esp++);
200         if (!(sa->sa_flags & SA_NOMASK))
201                 put_fs_long(current->blocked,tmp_esp++);
202         put_fs_long(eax,tmp_esp++);
203         put_fs_long(ecx,tmp_esp++);
204         put_fs_long(edx,tmp_esp++);
205         put_fs_long(eflags,tmp_esp++);
206         put_fs_long(old_eip,tmp_esp++);
207         current->blocked |= sa->sa_mask;   // 进程阻塞码(屏蔽码)添上 sa_mask 中的码位。
208         return(0);              /* Continue, execute handler */
209 }
210
```