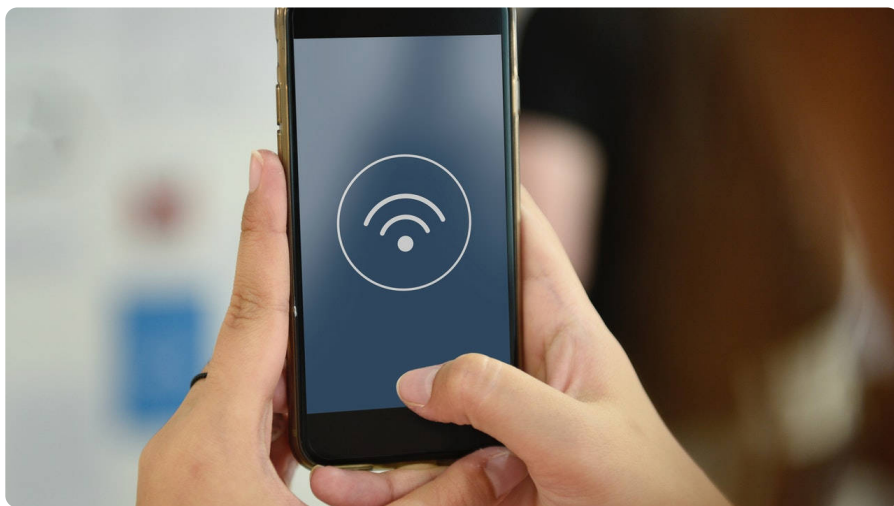


38 | 信号（下）：项目组A完成了，如何及时通知项目组B？

2019-06-24 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 23:30 大小 18.84M



信号处理最常见的流程主要是两步，第一步是注册信号处理函数，第二步是发送信号和处理信号。上一节，我们讲了注册信号处理函数，那一般什么情况下会产生信号呢？我们这一节就来看一看。

信号的发送

有时候，我们在终端输入某些组合键的时候，会给进程发送信号，例如，Ctrl+C 产生 SIGINT 信号，Ctrl+Z 产生 SIGTSTP 信号。

有的时候，硬件异常也会产生信号。比如，执行了除以 0 的指令，CPU 就会产生异常，然后把 SIGFPE 信号发送给进程。再如，进程访问了非法内存，内存管理模块就会产生异常，然后把信号 SIGSEGV 发送给进程。

这里同样是硬件产生的，对于中断和信号还是要加以区别。咱们前面讲过，中断要注册中断处理函数，但是中断处理函数是在内核驱动里面的，信号也要注册信号处理函数，信号处理函数是在用户态进程里面的。

对于硬件触发的，无论是中断，还是信号，肯定是先到内核的，然后内核对于中断和信号处理方式不同。一个是完全在内核里面处理完毕，一个是将信号放在对应的进程 task_struct 里信号相关的数据结构里面，然后等待进程在用户态去处理。当然有些严重的信号，内核会把进程干掉。但是，这也能看出来，中断和信号的严重程度不一样，信号影响的往往是某一个进程，处理慢了，甚至错了，也不过这个进程被干掉，而中断影响的是整个系统。一旦中断处理中有了 bug，可能整个 Linux 都挂了。

有时候，内核在某些情况下，也会给进程发送信号。例如，向读端已关闭的管道写数据时产生 SIGPIPE 信号，当子进程退出时，我们要给父进程发送 SIG_CHLD 信号等。

最直接的发送信号的方法就是，通过命令 kill 来发送信号了。例如，我们知道的 kill -9 pid 可以发送信号给一个进程，杀死它。

另外，我们还可以通过 kill 或者 sigqueue 系统调用，发送信号给某个进程，也可以通过 tkill 或者 tgkill 发送信号给某个线程。虽然方式多种多样，但是最终都是调用了 do_send_sig_info 函数，将信号放在相应的 task_struct 的信号数据结构中。


```
kill->kill_something_info->kill_pid_info-  
>group_send_sig_info->do_send_sig_info
```

```
tkill->do_tkill->do_send_specific->do_send_sig_info
```

```
tgkill->do_tkill->do_send_specific-  
>do_send_sig_info
```

```
rt_sigqueueinfo->do_rt_sigqueueinfo-  
>kill_proc_info->kill_pid_info->group_send_sig_info-  
>do_send_sig_info
```

do_send_sig_info 会调用 send_signal, 进而调用 __send_signal。

 复制代码

```
1 SYSCALL_DEFINE2(kill, pid_t, pid, int, sig)
2 {
3     struct siginfo info;
4
5     info.si_signo = sig;
6     info.si_errno = 0;
7     info.si_code = SI_USER;
8     info.si_pid = task_tgid_vnr(current);
9     info.si_uid = from_kuid_munged(current_user_ns(
10
11     return kill_something_info(sig, &info, pid);
12 }
13
14
15 static int __send_signal(int sig, struct siginfo *info,
16                          int group, int from_ancestor_ns
17 {
18     struct sigpending *pending;
19     struct sigqueue *q;
20     int override_rlimit;
21     int ret = 0, result;
22     .....
23     pending = group ? &t->signal->shared_pending :
24     .....
25     if (legacy_queue(pending, sig))
26         goto ret;
27
28     if (sig < SIGRTMIN)
29         override_rlimit = (is_si_special(info)
```

```

30     else
31         override_rlimit = 0;
32
33     q = __sigqueue_alloc(sig, t, GFP_ATOMIC | __GFP
34         override_rlimit);
35     if (q) {
36         list_add_tail(&q->list, &pending->list)
37         switch ((unsigned long) info) {
38             case (unsigned long) SEND_SIG_NOINFO:
39                 q->info.si_signo = sig;
40                 q->info.si_errno = 0;
41                 q->info.si_code = SI_USER;
42                 q->info.si_pid = task_tgid_nr_r
43
44                 q->info.si_uid = from_kuid_mung
45                 break;
46             case (unsigned long) SEND_SIG_PRIV:
47                 q->info.si_signo = sig;
48                 q->info.si_errno = 0;
49                 q->info.si_code = SI_KERNEL;
50                 q->info.si_pid = 0;
51                 q->info.si_uid = 0;
52                 break;
53             default:
54                 copy_siginfo(&q->info, info);
55                 if (from_ancestor_ns)
56                     q->info.si_pid = 0;
57                 break;
58         }
59
60         userns_fixup_signal_uid(&q->info, t);
61
62     }
63     .....
64     out_set:


```

```
65         signalfd_notify(t, sig);
66         sigaddset(&pending->signal, sig);
67         complete_signal(sig, t, group);
68 ret:
69         return ret;
70 }
```




在这里，我们看到，在学习进程数据结构中 `task_struct` 里面的 `sigpending`。在上面的代码里面，我们先是要决定应该用哪个 `sigpending`。这就要看我们发送的信号，是给进程的还是线程的。如果是 `kill` 发送的，也就是发送给整个进程的，就应该发送给 `t->signal->shared_pending`。这里面是整个进程所有线程共享的信号；如果是 `tkill` 发送的，也就是发给某个线程的，就应该发给 `t->pending`。这里面是这个线程的 `task_struct` 独享的。

`struct sigpending` 里面有两个成员，一个是一个集合 `sigset_t`，表示都收到了哪些信号，还有一个链表，也表示收到了哪些信号。它的结构如下：

 复制代码

```
1 struct sigpending {
2     struct list_head list;
3     sigset_t signal;
4 };
```

如果都表示收到了信号，这两者有什么区别呢？我们接着往下看 `__send_signal` 里面的代码。接下来，我们要调用 `legacy_queue`。如果满足条件，那就直接退出。那 `legacy_queue` 里面判断的是什么条件呢？我们来看它的代码。

 复制代码

```
1 static inline int legacy_queue(struct sigpending *signa
2 {
3     return (sig < SIGRTMIN) && sigismember(&signals
4 }
5
6
7 #define SIGRTMIN      32
8 #define SIGRTMAX      _NSIG
9 #define _NSIG         64
```

当信号小于 `SIGRTMIN`，也即 32 的时候，如果我们发现这个信号已经在集合里面了，就直接退出了。这样会造成什么现象呢？就是信号的丢失。例如，我们发送给进程 100 个 `SIGUSR1`（对应的信号为 10），那最终能够被我们的信号处理函数处理的信号有多少呢？这就不好说了，比如总共 5 个 `SIGUSR1`，分别是 A、B、C、D、E。


如果这五个信号来得太密。A 来了，但是信号处理函数还没来得及处理，B、C、D、E 就都来了。根据上面的逻辑，因为 A 已经将 SIGUSR1 放在 sigset_t 集合中了，因而后面四个都要丢失。如果是另一种情况，A 来了已经被信号处理函数处理了，内核在调用信号处理函数之前，我们会将集合中的标志位清除，这个时候 B 再来，B 还是会进入集合，还是会被处理，也就不会丢。

这样信号能够处理多少，和信号处理函数什么时候被调用，信号多大频率被发送，都有关系，而且从后面的分析，我们可以知道，信号处理函数的调用时间也是不确定的。看小于 32 的信号如此不靠谱，我们就称它为**不可靠信号**。

如果大于 32 的信号是什么情况呢？我们接着看。接下来，__sigqueue_alloc 会分配一个 struct sigqueue 对象，然后通过 list_add_tail 挂在 struct sigpending 里面的链表上。这样就靠谱多了是不是？如果发送过来 100 个信号，变成链表上的 100 项，都不会丢，哪怕相同的信号发送多遍，也处理多遍。因此，大于 32 的信号我们称为**可靠信号**。当然，队列的长度也是有限制的，如果我们执行 ulimit 命令，可以看到，这个限制 pending signals (-i) 15408。

当信号挂到了 task_struct 结构之后，最后我们需要调用 complete_signal。这里面的逻辑也很简单，就是说，既然


这个进程有了一个新的信号，赶紧找一个线程处理一下吧。

 复制代码

```
1 static void complete_signal(int sig, struct task_struct
2 {
3     struct signal_struct *signal = p->signal;
4     struct task_struct *t;
5
6     /*
7      * Now find a thread we can wake up to take the
8      *
9      * If the main thread wants the signal, it gets
10     * Probably the least surprising to the average
11     */
12     if (wants_signal(sig, p))
13         t = p;
14     else if (!group || thread_group_empty(p))
15         /*
16          * There is just one thread and it does
17          * It will dequeue unblocked signals be
18          */
19         return;
20     else {
21         /*
22          * Otherwise try to find a suitable thr
23          */
24         t = signal->curr_target;
25         while (!wants_signal(sig, t)) {
26             t = next_thread(t);
27             if (t == signal->curr_target)
28                 return;
29         }
30         signal->curr_target = t;
31     }
```

```
32 .....
33     /*
34     * The signal is already in the shared-pending
35     * Tell the chosen thread to wake up and dequeu
36     */
37     signal_wake_up(t, sig == SIGKILL);
38     return;
39 }
```

在找到了一个进程或者线程的 `task_struct` 之后，我们要调用 `signal_wake_up`，来企图唤醒它，`signal_wake_up` 会调用 `signal_wake_up_state`。

 复制代码

```
1 void signal_wake_up_state(struct task_struct *t, unsig
2 {
3     set_tsk_thread_flag(t, TIF_SIGPENDING);
4
5
6     if (!wake_up_state(t, state | TASK_INTERRUPTIBL
7         kick_process(t);
8 }
```

`signal_wake_up_state` 里面主要做了两件事情。第一，就是给这个线程设置 `TIF_SIGPENDING`，这就说明其实信号

的处理和进程的调度是采取这样一种类似的机制。还记得咱们调度的时候是怎么操作的吗？


当发现一个进程应该被调度的时候，我们并不直接把它赶下来，而是设置一个标识位 `TIF_NEED_RESCHED`，表示等待调度，然后等待系统调用结束或者中断处理结束，从内核态返回用户态的时候，调用 `schedule` 函数进行调度。信号也是类似的，当信号来的时候，我们并不直接处理这个信号，而是设置一个标识位 `TIF_SIGPENDING`，来表示已经有信号等待处理。同样等待系统调用结束，或者中断处理结束，从内核态返回用户态的时候，再进行信号的处理。

`signal_wake_up_state` 的第二件事情，就是试图唤醒这个进程或者线程。`wake_up_state` 会调用 `try_to_wake_up` 方法。这个函数我们讲进程的时候讲过，就是将这个进程或者线程设置为 `TASK_RUNNING`，然后放在运行队列中，这个时候，当随着时钟不断的滴答，迟早会被调用。如果 `wake_up_state` 返回 0，说明进程或者线程已经是 `TASK_RUNNING` 状态了，如果它在另外一个 CPU 上运行，则调用 `kick_process` 发送一个处理器间中断，强制那个进程或者线程重新调度，重新调度完毕后，会返回用户态运行。这是一个时机检查 `TIF_SIGPENDING` 标识位。

信号的处理

好了，信号已经发送到位了，什么时候真正处理它呢？

就是在从系统调用或者中断返回的时候，咱们讲调度的时候讲过，无论是从系统调用返回还是从中断返回，都会调用 `exit_to_usermode_loop`，只不过我们上次主要关注了 `_TIF_NEED_RESCHED` 这个标识位，这次我们重点关注 **`_TIF_SIGPENDING` 标识位**。

 复制代码

```
1 static void exit_to_usermode_loop(struct pt_regs *regs,
2 {
3     while (true) {
4         .....
5         if (cached_flags & _TIF_NEED_RESCHED)
6             schedule();
7         .....
8         /* deal with pending signal delivery */
9         if (cached_flags & _TIF_SIGPENDING)
10            do_signal(regs);
11        .....
12        if (!(cached_flags & EXIT_TO_USERMODE_L
13            break;
14    }
15 }
```


如果在前一个环节中，已经设置了 `_TIF_SIGPENDING`，我们就调用 `do_signal` 进行处理。

```
1 void do_signal(struct pt_regs *regs)
2 {
3     struct ksignal ksig;
4
5     if (get_signal(&ksig)) {
6         /* Whee! Actually deliver the signal.
7          handle_signal(&ksig, regs);
8          return;
9     }
10
11     /* Did we come from a system call? */
12     if (syscall_get_nr(current, regs) >= 0) {
13         /* Restart the system call - no handler
14          switch (syscall_get_error(current, regs)
15          case -ERESTARTNOHAND:
16          case -ERESTARTSYS:
17          case -ERESTARTNOINTR:
18              regs->ax = regs->orig_ax;
19              regs->ip -= 2;
20              break;
21
22          case -ERESTART_RESTARTBLOCK:
23              regs->ax = get_nr_restart_sysca
24              regs->ip -= 2;
25              break;
26          }
27     }
28     restore_saved_sigmask();
29 }
```



do_signal 会调用 handle_signal。按说，信号处理就是调用用户提供的信号处理函数，但是这事儿没有看起来这么简单，因为信号处理函数是在用户态的。

咱们又要来回忆系统调用的过程了。这个进程当时在用户态执行到某一行 Line A，调用了一个系统调用，在进入内核的那一刻，在内核 pt_regs 里面保存了用户态执行到了 Line A。现在我们从系统调用返回用户态了，按说应该从 pt_regs 拿出 Line A，然后接着 Line A 执行下去，但是为了响应信号，我们不能回到用户态的时候返回 Line A 了，而是应该返回信号处理函数的起始地址。

 复制代码

```
1 static void
2 handle_signal(struct ksignal *ksig, struct pt_regs *regs
3 {
4     bool stepping, failed;
5     .....
6     /* Are we from a system call? */
7     if (syscall_get_nr(current, regs) >= 0) {
8         /* If so, check system call restarting.
9         switch (syscall_get_error(current, regs
10         case -ERESTART_RESTARTBLOCK:
11         case -ERESTARTNOHAND:
12             regs->ax = -EINTR;
13             break;
14         case -ERESTARTSYS:
15             if (!(ksig->ka.sa.sa_flags & SA
16                 regs->ax = -EINTR;
```

```

17                                     break;
18                                 }
19                             /* fallthrough */
20                             case -ERESTARTNOINTR:
21                                 regs->ax = regs->orig_ax;
22                                 regs->ip -= 2;
23                                 break;
24                             }
25                     }
26     .....
27     failed = (setup_rt_frame(ksig, regs) < 0);
28     .....
29     signal_setup_done(failed, ksig, stepping);
30 }

```



这个时候，我们就需要干预和自己来定制 `pt_regs` 了。这个时候，我们要看，是否从系统调用中返回。如果是从系统调用返回的话，还要区分我们是从系统调用中正常返回，还是在一个非运行状态的系统调用中，因为会被信号中断而返回。

我们这里解析一个最复杂的场景。还记得咱们解析进程调度的时候，我们举的一个例子，就是从一个 `tap` 网卡中读取数据。当时我们主要关注 `schedule` 那一行，也即如果当发现没有数据的时候，就调用 `schedule`，自己进入等待状态，然后将 CPU 让给其他进程。具体的代码如下：

```
1 static ssize_t tap_do_read(struct tap_queue *q,
2                             struct iov_iter *to,
3                             int noblock, struct sk_buff
4 {
5     .....
6     while (1) {
7         if (!noblock)
8             prepare_to_wait(sk_sleep(&q->sk
9                             TASK_INTERRUPTI
10
11         /* Read frames from the queue */
12         skb = skb_array_consume(&q->skb_array);
13         if (skb)
14             break;
15         if (noblock) {
16             ret = -EAGAIN;
17             break;
18         }
19         if (signal_pending(current)) {
20             ret = -ERESTARTSYS;
21             break;
22         }
23         /* Nothing to read, let's sleep */
24         schedule();
25     }
26     .....
27 }
```

这里我们关注和信号相关的部分。这其实是一个信号中断系统调用的典型逻辑。


首先，我们把当前进程或者线程的状态设置为 `TASK_INTERRUPTIBLE`，这样才能是使这个系统调用可以被中断。

其次，可以被中断的系统调用往往是比较慢的调用，并且会因为数据不就绪而通过 `schedule` 让出 CPU 进入等待状态。在发送信号的时候，我们除了设置这个进程和线程的 `_TIF_SIGPENDING` 标识位之外，还试图唤醒这个进程或者线程，也就是将它从等待状态中设置为 `TASK_RUNNING`。

当这个进程或者线程再次运行的时候，我们根据进程调度第一定律，从 `schedule` 函数中返回，然后再次进入 `while` 循环。由于这个进程或者线程是由信号唤醒的，而不是因为数据来了而唤醒的，因而是读不到数据的，但是在 `signal_pending` 函数中，我们检测到了 `_TIF_SIGPENDING` 标识位，这说明系统调用没有真的做完，于是返回一个错误 `ERESTARTSYS`，然后带着这个错误从系统调用返回。

然后，我们到了 `exit_to_usermode_loop->do_signal->handle_signal`。在这里面，当发现出现错误 `ERESTARTSYS` 的时候，我们就知道这是从一个没有调用完的系统调用返回的，设置系统调用错误码 `EINTR`。

接下来，我们就开始折腾 pt_regs 了，主要通过调用 setup_rt_frame->__setup_rt_frame。

 复制代码

```
1 static int __setup_rt_frame(int sig, struct ksignal *ks
2                               sigset_t *set, struct pt_re
3 {
4     struct rt_sigframe __user *frame;
5     void __user *fp = NULL;
6     int err = 0;
7
8     frame = get_sigframe(&ksig->ka, regs, sizeof(st
9     .....
10    put_user_try {
11    .....
12                /* Set up to return from userspace.  If
13                   already in userspace.  */
14                /* x86-64 should always use SA_RESTOREF
15                   if (ksig->ka.sa.sa_flags & SA_RESTORER)
16                       put_user_ex(ksig->ka.sa.sa_rest
17                   }
18    } put_user_catch(err);
19
20    err |= setup_sigcontext(&frame->uc.uc_mcontext,
21    err |= __copy_to_user(&frame->uc.uc_sigmask, se
22
23    /* Set up registers for signal handler */
24    regs->di = sig;
25    /* In case the signal handler was declared with
26    regs->ax = 0;
27
28    regs->si = (unsigned long)&frame->info;
29    regs->dx = (unsigned long)&frame->uc;
```

```
30         regs->ip = (unsigned long) ksig->ka.sa.sa_handler
31
32         regs->sp = (unsigned long)frame;
33         regs->cs = __USER_CS;
34         .....
35         return 0;
36     }
```



frame 的类型是 `rt_sigframe`。frame 的意思是帧。我们只有在学习栈的时候，提到过栈帧的概念。对的，这个 frame 就是一个栈帧。

我们在 `get_sigframe` 中会得到 `pt_regs` 的 `sp` 变量，也就是原来这个程序在用户态的栈顶指针，然后 `get_sigframe` 中，我们会将 `sp` 减去 `sizeof(struct rt_sigframe)`，也就是把这个栈帧塞到了栈里面，然后我们又在 `__setup_rt_frame` 中把 `regs->sp` 设置成等于 `frame`。这就相当于强行在程序原来的用户态的栈里面插入了一个栈帧，并在最后将 `regs->ip` 设置为用户定义的信号处理函数 `sa_handler`。这意味着，本来返回用户态应该接着原来的代码执行的，现在不了，要执行 `sa_handler` 了。那执行完了以后呢？按照函数栈的规则，弹出上一个栈帧来，也就是弹出了 `frame`。


那如果我们假设 `sa_handler` 成功返回了，怎么回到程序原来在用户态运行的地方呢？玄机就在 `frame` 里面。要想恢

复原来运行的地方，首先，原来的 `pt_regs` 不能丢，这个问题，是在 `setup_sigcontext` 里面，将原来的 `pt_regs` 保存在了 `frame` 中的 `uc_mcontext` 里面。

另外，很重要的一点，程序如何跳过去呢？在 `__setup_rt_frame` 中，还有一个不引起重视的操作，那就是通过 `put_user_ex`，将 `sa_restorer` 放到了 `frame->precode` 里面，而且还是按照函数栈的规则。函数栈里面包含了函数执行完跳回去的地址。当 `sa_handler` 执行完之后，弹出的函数栈是 `frame`，也就应该跳到 `sa_restorer` 的地址。这是什么地址呢？

咱们在 `sigaction` 介绍的时候就没有介绍它，在 Glibc 的 `__libc_sigaction` 函数中也没有注意到，它被赋值成了 `restore_rt`。这其实就是 `sa_handler` 执行完毕之后，马上要执行的函数。从名字我们就能感觉到，它将恢复原来程序运行的地方。

在 Glibc 中，我们可以找到它的定义，它竟然调用了一个系统调用，系统调用号为 `__NR_rt_sigreturn`。

 复制代码

```
1 RESTORE (restore_rt, __NR_rt_sigreturn)
2
3 #define RESTORE(name, syscall) RESTORE2 (name, syscall)
```


```

4 # define RESTORE2(name, syscall) \
5 asm                                     \
6 (                                     \
7     ".LSTART_" #name ":\n"          \
8     "    .type __" #name ",@function\n" \
9     "    __" #name ":\n"              \
10    "    movq $" #syscall ", %rax\n"   \
11    "    syscall\n"                    \
12    .....

```



我们可以在内核里面找到 `__NR_rt_sigreturn` 对应的系统调用。

 复制代码

```

1 asmlinkage long sys_rt_sigreturn(void)
2 {
3     struct pt_regs *regs = current_pt_regs();
4     struct rt_sigframe __user *frame;
5     sigset_t set;
6     unsigned long uc_flags;
7
8     frame = (struct rt_sigframe __user *) (regs->sp
9     if (__copy_from_user(&set, &frame->uc.uc_sigmas
10         goto badframe;
11     if (__get_user(uc_flags, &frame->uc.uc_flags))
12         goto badframe;
13
14     set_current_blocked(&set);
15
16     if (restore_sigcontext(regs, &frame->uc.uc_mcor

```

```
17             goto badframe;
18 .....
19         return regs->ax;
20 .....
21 }
```



在这里面，我们把上次填充的那个 `rt_sigframe` 拿出来，然后 `restore_sigcontext` 将 `pt_regs` 恢复成为原来用户态的样子。从这个系统调用返回的时候，应用还误以为从上次的系统调用返回的呢。

至此，整个信号处理过程才全部结束。

总结时刻

信号的发送与处理是一个复杂的过程，这里来总结一下。

1. 假设我们有一个进程 A，`main` 函数里面调用系统调用进入内核。
2. 按照系统调用的原理，会将用户态栈的信息保存在 `pt_regs` 里面，也即记住原来用户态是运行到了 line A 的地方。
3. 在内核中执行系统调用读取数据。

4. 当发现没有什么数据可读取的时候，只好进入睡眠状态，并且调用 `schedule` 让出 CPU，这是进程调度第一定律。
5. 将进程状态设置为 `TASK_INTERRUPTIBLE`，可中断的睡眠状态，也即如果有信号来的话，是可以唤醒它的。
6. 其他的进程或者 shell 发送一个信号，有四个函数可以调用 `kill`, `tkill`, `tgkill`, `rt_sigqueueinfo`
7. 四个发送信号的函数，在内核中最终都是调用 `do_send_sig_info`
8. `do_send_sig_info` 调用 `send_signal` 给进程 A 发送一个信号，其实就是找到进程 A 的 `task_struct`，或者加入信号集合，为不可靠信号，或者加入信号链表，为可靠信号
9. `do_send_sig_info` 调用 `signal_wake_up` 唤醒进程 A。
10. 进程 A 重新进入运行状态 `TASK_RUNNING`，根据进程调度第一定律，一定会接着 `schedule` 运行。
11. 进程 A 被唤醒后，检查是否有信号到来，如果没有，重新循环到一开始，尝试再次读取数据，如果还是没有数据，再次进入 `TASK_INTERRUPTIBLE`，即可中断的睡眠状态。
12. 当发现有信号到来的时候，就返回当前正在执行的系统调用，并返回一个错误表示系统调用被中断了。
13. 系统调用返回的时候，会调用 `exit_to_usermode_loop`，这是一个处理信号的时机
14. 调用 `do_signal` 开始处理信号

15. 根据信号，得到信号处理函数 `sa_handler`，然后修改 `pt_regs` 中的用户态栈的信息，让 `pt_regs` 指向 `sa_handler`。同时修改用户态的栈，插入一个栈帧 `sa_restorer`，里面保存了原来的指向 line A 的 `pt_regs`，并且设置让 `sa_handler` 运行完毕后，跳到 `sa_restorer` 运行。
16. 返回用户态，由于 `pt_regs` 已经设置为 `sa_handler`，则返回用户态执行 `sa_handler`。
17. `sa_handler` 执行完毕后，信号处理函数就执行完了，接着根据第 15 步对于用户态栈帧的修改，会跳到 `sa_restorer` 运行。
18. `sa_restorer` 会调用系统调用 `rt_sigreturn` 再次进入内核。
19. 在内核中，`rt_sigreturn` 恢复原来的 `pt_regs`，重新指向 line A。
20. 从 `rt_sigreturn` 返回用户态，还是调用 `exit_to_usermode_loop`。
21. 这次因为 `pt_regs` 已经指向 line A 了，于是就到了进程 A 中，接着系统调用之后运行，当然这个系统调用返回的是它被中断了，没有执行完的错误。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 37 | 信号（上）：项目组A完成了，如何及时通知项目...

精选留言 (5)

💬 写留言



安排

2019-06-24

能把这个流程串起来，老师功力深厚啊



👍 2



许童童

2019-06-24

讲得太好了，真是深入浅出啊！



Sharry

2019-06-24

Linux 信号通信主要由如下几个步骤组成

- 信号处理函数的注册
 - 信号处理函数的注册, 定义在用户空间
 - 注册最终通过 `rt_sigaction` 系统调用发起
 - 将用户空间定义的信号处理函数保存到 `task_struct` ...



刘強

2019-06-24

老师，既然内核态对用户态的栈随意操作（果然是内核，权利就是大），但返回的时候还是保持系统调用前的样子，丝毫没有察觉背后发生了这么多事情，就好像调用了一个普通用户态的函数一样，那么我在用户态调试程序的时候，能否看到这种内核对用户栈的修改？



小龙的城堡

2019-06-24

这一章讲得非常棒！很清晰！感谢老师！

