Linux 信号

信号本质上是整数，用户模式下用来模拟硬件中断

什么是硬件中断？先理解为硬件引发的中断。什么是中断？简言之，就是让CPU停下当前干的事转而去处理新的情况。

信号是谁产生的？可以由硬件、内核和进程产生。

例如在终端上用Ctrl+C，可以结束掉当前终端进程，本质上是发送SIGINT信号

如下是一个实例程序，该程序循环打印数字。运行起来，看看Ctrl+C能不能中止它

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdbool.h>

void hand(int val)
{
    printf("val: %d\n",val);
    printf("Do you want to kill me? No way!\n");
}

int main()
{
    signal(SIGINT,hand);
    int n = 0;
    while (true) {
        printf("n:%d\n",n++);
        sleep(1);
    }
    return 0;
}
```

如下所示，现在使用Ctrl+C是没办法中断这个程序的

```
hwx@N3ptune:~/cprogram/testc$ gcc demo.c -o demo
hwx@N3ptune:~/cprogram/testc$ ./demo
n:0
n:1
n:2
n:3
^Cval: 2
Do you want to kill me? No way!
n:4
n:5
n:6
^Cval: 2
Do you want to kill me? No way!
n:7
n:8
n:9
n:10
```

这里可以用man命令查看一下signal函数的帮助文档

```
NAME
       signal - ANSI C signal handling

SYNOPSIS
       #include <signal.h>

       typedef void (*sighandler_t)(int);

       sighandler_t signal(int signum, sighandler_t handler);

DESCRIPTION
       The  behavior  of  signal()  varies  across UNIX versions, and has also varied historically across different versions of
       Linux.  Avoid its use: use sigaction(2) instead.  See Portability below.

       signal() sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or  the  address  of  a
       programmer-defined function (a "signal handler").
```

signal关联上了一个信号处理函数，来处理信号

现在修改这个代码，添加一行打印当前进程号，然后打开两个终端，一个终端运行上面的程序，一个终端尝试kill命令来发送信号

```
val: 2
Do you want to kill me? No way!
n:25
n:26
n:27
n:28
val: 2
Do you want to kill me? No way!
n:29
val: 2
Do you want to kill me? No way!
```

kill 要指定参数

```
hwx@N3ptune:~/cprogram/testc$ kill -s 2 19233
hwx@N3ptune:~/cprogram/testc$ kill -s 2 19233
hwx@N3ptune:~/cprogram/testc$ kill -s 2 19233
hwx@N3ptune:~/cprogram/testc$
```

这是可行的，结果如上

信号处理的过程

进程A在运行，内核、硬件或者其他进程发送信号给进程A。进程A接收到信号后，直接做信号本身规定的对应处理或者做事先注册好的信号处理。如上面signal函数就是注册信号处理，hand函数替换了本身默认的信号处理。当然信号是可以屏蔽的，不作任何处理。

Linux有哪些信号类型

```
hwx@N3ptune:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Linux 信号有64个，分为不可靠信号(非实时，1-31，Unix提供)和可靠信号(32-64，后来扩充的)。又有系统自带的标准信号和用户自定义的信号。

介绍几个命令或函数

信号注册：signal、sigaction

信号发送：kill命令、kill函数、sigqueue

信号屏蔽：sigprocmask

信号集：sigset_t

下面写一个发送信号的程序

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
    int pid = atoi(argv[1]);
    int sig = atoi(argv[2]);

    printf("pid:%d  sig:%d\n",pid,sig);
    kill(pid,sig);
    return 0;
}
```

```
val: 2
Do you want to kill me? No way!
n:24
n:25
```

```
hwx@N3ptune:~/cprogram/testc$ ./demo1 24523 2
pid:24523  sig:2
```

成功发送了信号

sigaction函数有所不同，功能更多

```
NAME
       sigaction, rt_sigaction - examine and change a signal action

SYNOPSIS
       #include <signal.h>

       int sigaction(int signum, const struct sigaction *act,
                     struct sigaction *oldact);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       sigaction(): _POSIX_C_SOURCE

       siginfo_t: _POSIX_C_SOURCE >= 199309L

DESCRIPTION
       The  sigaction()  system  call  is used to change the action taken by a process on receipt of a specific signal.  (See
       signal(7) for an overview of signals.)

       signum specifies the signal and can be any valid signal except SIGKILL and SIGSTOP.
```

这里用到了一个结构体

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

结构体的第一个成员依然是原来的信号处理函数，第二个成员被称作高级信号处理函数，第三个成员用于信号屏蔽，剩下两个暂时还用不到。

这个函数不仅可以发信号、做信号处理，还可以接收信号的同时接收数据

下列代码使用了高级信号处理

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdbool.h>

// 基本信号处理函数
void hand(int n)
{
    printf("基本信号处理函数\n");
}

// 高级信号处理函数
void handler(int n,siginfo_t* siginfo,void* arg)
{
    printf("高级信号处理函数\n");
    printf("n:%d msg:%d\n",n,siginfo->si_int);
}

int main(void)
{
    struct sigaction act = {0};
    struct sigaction old_act = {0};
    act.sa_handler = hand;
    act.sa_sigaction = handler;
    act.sa_flags = SA_SIGINFO;

    // 高级信号处理
    sigaction(2,&act,&old_act);
    printf("pid:%d\n",getpid());

    int n = 0;
    while (true) {
        printf("n:%d\n",n++);
        sleep(1);
    }
    return 0;
}
```

同时还要实现高级的信号发送，采用sigqueue

```
NAME
       sigqueue - queue a signal and data to a process

SYNOPSIS
       #include <signal.h>

       int sigqueue(pid_t pid, int sig, const union sigval value);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       sigqueue(): _POSIX_C_SOURCE >= 199309L

DESCRIPTION
       sigqueue()  sends  the  signal specified in sig to the process whose PID is given in pid.  The permissions required to
       send a signal are the same as for kill(2).  As with kill(2), the null signal (0) can be used to  check  if  a  process
       with a given PID exists.
```

这里有个重要的联合体，用来存储数据和信号一起发送

```
   The  value  argument is used to specify an accompanying item of data (either an integer or a pointer value) to be sent
   with the signal, and has the following type:

       union sigval {
           int   sival_int;
           void *sival_ptr;
       };
```

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
    int pid = atoi(argv[1]);
    int sig = atoi(argv[2]);
    printf("pid:%d  sig:%d\n",argv[1],argv[2]);
    union sigval u;
    u.sival_int = 12345678;
    sigqueue(pid,sig,u);
    return 0;
}
```

运行结果如下

```
hwx@N3ptune:~/cprogram/testc$ ./sigqueue 31560 2
pid:-829645121  sig:-829645115
```

```
pid:31560
n:0
n:1
n:2
n:3
n:4
n:5
n:6
n:7
高级信号处理函数
n:2 msg:12345678
n:8
n:9
n:10
n:11
```

可以看到程序接收到了信号和数据

也试试发送其他数据，联合体中还有一个指针类型的成员

Linux信号(续)

参考网站: Github、《极客时间》

现在从底层角度来观赏Linux信号机制

首先看信号的具体作用

```
Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP       1       Term    Hangup detected on controlling terminal
                             or death of controlling process
SIGINT       2       Term    Interrupt from keyboard
SIGQUIT      3       Core    Quit from keyboard
SIGILL       4       Core    Illegal Instruction
SIGABRT      6       Core    Abort signal from abort(3)
SIGFPE       8       Core    Floating-point exception
SIGKILL      9       Term    Kill signal
SIGSEGV     11       Core    Invalid memory reference
SIGPIPE     13       Term    Broken pipe: write to pipe with no
                             readers; see pipe(7)
SIGALRM     14       Term    Timer signal from alarm(2)
SIGTERM     15       Term    Termination signal
SIGUSR1   30,10,16   Term    User-defined signal 1
SIGUSR2   31,12,17   Term    User-defined signal 2
SIGCHLD   20,17,18   Ign     Child stopped or terminated
SIGCONT   19,18,25   Cont    Continue if stopped
SIGSTOP   17,19,23   Stop    Stop process
SIGTSTP   18,20,24   Stop    Stop typed at terminal
SIGTTIN   21,21,26   Stop    Terminal input for background process
SIGTTOU   22,22,27   Stop    Terminal output for background process
```

```
Next the signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.

Signal       Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGBUS      10,7,10     Core    Bus error (bad memory access)
SIGPOLL                 Term    Pollable event (Sys V).
                                Synonym for SIGIO
SIGPROF     27,27,29    Term    Profiling timer expired
SIGSYS      12,31,12    Core    Bad system call (SVr4);
                                see also seccomp(2)
SIGTRAP        5        Core    Trace/breakpoint trap
SIGURG      16,23,21    Ign     Urgent condition on socket (4.2BSD)
SIGVTALRM   26,26,28    Term    Virtual alarm clock (4.2BSD)
SIGXCPU     24,24,30    Core    CPU time limit exceeded (4.2BSD);
                                see setrlimit(2)
SIGXFSZ     25,25,31    Core    File size limit exceeded (4.2BSD);
                                see setrlimit(2)
```

如上文所提，处理信号有3种方式：执行默认操作，捕获信号和忽略信号

设置信号处理有两种方式: signal和sigaction

signal将一个动作和信号关联，sigaction也是将一个动作和信号关联，但这个动作用一个结构体表示了，处理信号将更加细致

siganal不是系统调用，而是glibc封装的一个函数，实现如下：

```
#  define signal __sysv_signal
__sighandler_t
__sysv_signal (int sig, __sighandler_t handler)
{
  struct sigaction act, oact;
......
  act.sa_handler = handler;
  __sigemptyset (&act.sa_mask);
  act.sa_flags = SA_ONESHOT | SA_NOMASK | SA_INTERRUPT;
  act.sa_flags &= ~SA_RESTART;
  if (__sigaction (sig, &act, &oact) < 0)
    return SIG_ERR;
  return oact.sa_handler;
}
weak_alias (__sysv_signal, sysv_signal)
```

这里可以看到sa_flags 设置了一个SA_ONESHOT，这意味着信号处理函数只作用一次，用完一次后就返回默认行为

同时设置了SA_NOMASK，通过 __sigemptyset，将 sa_mask 设置为空。这样的设置表示在这个信号处理函数执行过程中，如果再有其他信号，哪怕相同的信号到来的时候，这个信号处理函数会被中断。如果一个信号处理函数真的被其他信号中断，问题也不大，因为当处理完了其他的信号处理函数后，还会回来接着处理这个信号处理函数的，但是对于相同的信号就有点尴尬了。一般的思路应该是，当某一个信号的信号处理函数运行的时候，暂时屏蔽这个信号，屏蔽并不意味着信号一定丢失，而是暂存，这样能够做到信号处理函数对于相同的信号，处理完一个再处理下一个，这样信号处理函数的逻辑要简单得多。

还有一个设置就是设置了 SA_INTERRUPT，清除了 SA_RESTART。信号的到来时间是不可预期的，有可能程序正在调用某个漫长的系统调用的时候，突然到来一个信号，中断了这个系统调用，去执行信号处理函数。那么执行完后信号处理函数，系统调用怎么办？

时候有两种处理方法，一种就是 SA_INTERRUPT，也即系统调用被中断了，就不再重试这个系统调用了，而是直接返回一个 -EINTR 常量，告诉调用方，这个系统调用被信号中断了，调用方可以根据自己的逻辑，重新调用或者直接返回，这会使得代码非常复杂，在所有系统调用的返回值判断里面，都要特殊判断一下这个值。

另外一种处理方法是 SA_RESTART。这个时候系统调用会被自动重新启动，不需要调用方自己写代码。当然也可能存在问题，例如从终端读入一个字符，这个时候用户在终端输入一个'a'字符，在处理'a'字符的时候被信号中断了，等信号处理完毕，再次读入一个字符的时候，如果用户不再输入，就停在那里了，需要用户再次输入同一个字符。

可知signal函数是受到限制的，因此，建议使用 sigaction 函数，根据自己的需要定制参数。

下面是glibc中的实现:

```
int
__sigaction (int sig, const struct sigaction *act, struct sigaction *oact)
{
......
  return __libc_sigaction (sig, act, oact);
}


int
__libc_sigaction (int sig, const struct sigaction *act, struct sigaction *oact)
{
  int result;
  struct kernel_sigaction kact, koact;


  if (act)
    {
      kact.k_sa_handler = act->sa_handler;
      memcpy (&kact.sa_mask, &act->sa_mask, sizeof (sigset_t));
      kact.sa_flags = act->sa_flags | SA_RESTORER;


      kact.sa_restorer = &restore_rt;
    }


  result = INLINE_SYSCALL (rt_sigaction, 4,
                           sig, act ? &kact : NULL,
                           oact ? &koact : NULL, _NSIG / 8);
  if (oact && result >= 0)
    {
      oact->sa_handler = koact.k_sa_handler;
      memcpy (&oact->sa_mask, &koact.sa_mask, sizeof (sigset_t));
      oact->sa_flags = koact.sa_flags;
      oact->sa_restorer = koact.sa_restorer;
    }
  return result;
}
```

内核代码注释表明，系统调用 signal 是为了兼容过去，系统调用 sigaction 也是为了兼容过去，连参数都变成了 struct compat_old_sigaction，所以说，我们的库函数虽然调用的是 sigaction，到了系统调用层，调用的可不是系统调用 sigaction，而是系统调用 rt_sigaction。

在 rt_sigaction 里面，将用户态的 struct sigaction 结构，拷贝为内核态的 k_sigaction，然后调用 do_sigaction。do_sigaction 也很简单，进程内核的数据结构里，struct task_struct 里面有一个成员 sighand，里面有一个 action。这是一个数组，下标是信号，内容就是信号处理函数，do_sigaction 就是设置 sighand 里的信号处理函数。

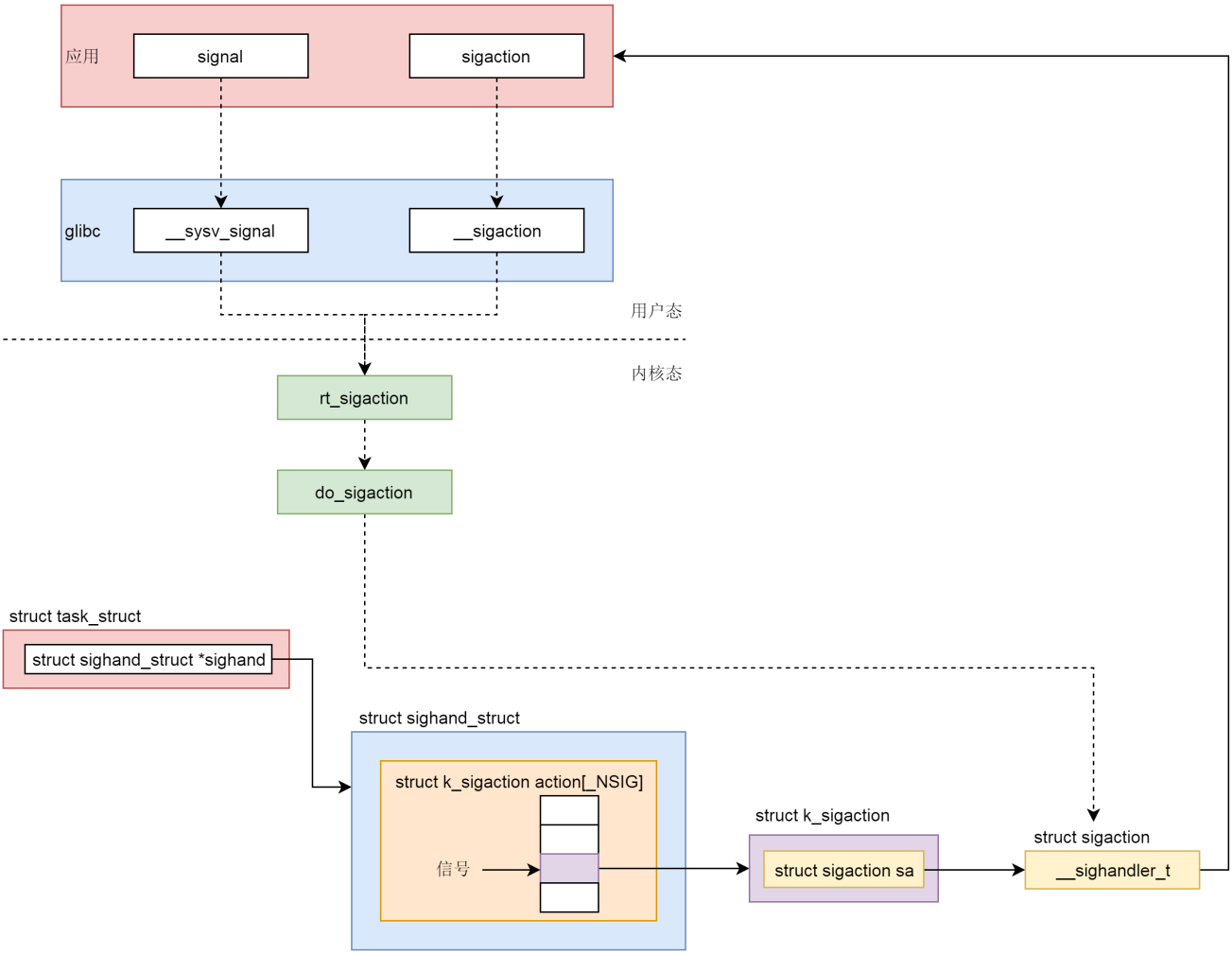
```
int do_sigaction(int sig, struct k_sigaction *act, struct k_sigaction *oact)
{
  struct task_struct *p = current, *t;
  struct k_sigaction *k;
  sigset_t mask;
......
  k = &p->sighand->action[sig-1];


  spin_lock_irq(&p->sighand->siglock);
  if (oact)
    *oact = *k;


  if (act) {
    sigdelsetmask(&act->sa.sa_mask,
            sigmask(SIGKILL) | sigmask(SIGSTOP));
    *k = *act;
......
  }


  spin_unlock_irq(&p->sighand->siglock);
  return 0;
}
```

总结:


应用
signal
sigaction
glibc
__sysv_signal
__sigaction
用户态
内核态
rt_sigaction
do_sigaction
struct task_struct
struct sighand_struct *sighand
struct sighand_struct
struct k_sigaction action[_NSIG]
信号
struct k_sigaction
struct sigaction sa
struct sigaction
__sighandler_t