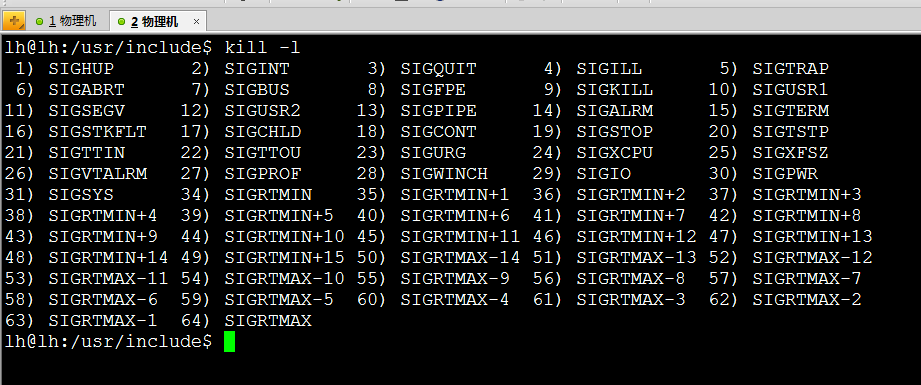
# 什么是信号？

信号是[软件中断](http://baike.baidu.com/link?url=474nN303T2Oe2ehYZjkrggeXCaJPDSrmM5Unoh4TTuty4wSgS0nl4-vl43AGMFbo0_5uH5OQFr_vaRJaZ-3lq_)，进程收到信号后会中断当前进程去处理中断。

Linux 可使用命令：kill -l（"l" 为字母）,查看系统拥有的信号。



编号为 1 ~ 31是不可靠信号（非实时的）：多次发送只会收到一次信号

编号为32 ~ 63是可靠信号（实时信号）：多次发送会收到多次信号

非可靠信号一般都有确定的用途及含义,  可靠信号则可以让用户自定义使用。

# 信号的产生方式

1）当用户按某些终端键时：如：终端上按“Ctrl+c”产生中断信号 SIGINT，编号2  
2）硬件异常将产生信号。  
3）软件异常将产生信号。  
 当检测到某种软件条件已发生，并将其通知有关进程时，产生信号。  
4）调用 kill() 函数将发送信号。  
 接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者是超级用户  
5）运行 kill 命令将发送信号。  
 此程序实际上是使用 kill 函数来发送信号。

# 信号的常用操作

## 发送信号

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int signum);

功能：

给指定进程发送信号。

接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者是超级用户

参数：

pid: 取值有 4 种情况:

pid > 0: 将信号传送给进程 ID 为pid的进程。

pid = 0: 将信号传送给当前进程所在进程组中的所有进程。

pid = -1: 将信号传送给系统内所有的进程。

pid < -1: 将信号传给指定进程组的所有进程。进程组号是 pid 的绝对值

signum: 信号的编号，这里可以填数字编号，也可以填信号的宏定义

返回值：

成功：0

失败：-1

下面为测试代码

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <signal.h>

int main(int argc, char \*argv[])

{

    pid\_t pid;

    int i = 0;

    pid = fork(); // 创建进程

    if( pid < 0 ){ // 出错

        perror("fork");

    }

    if(pid == 0){ // 子进程

        while(1){

            printf("I am son\n");

            sleep(1);

        }

    }else if(pid > 0){ // 父进程

        while(1){

            printf("I am father\n");

            sleep(1);

            i++;

            if(3 == i){// 3秒后

                kill(pid, SIGINT); // 给子进程 pid ，发送中断信号 SIGINT

            }

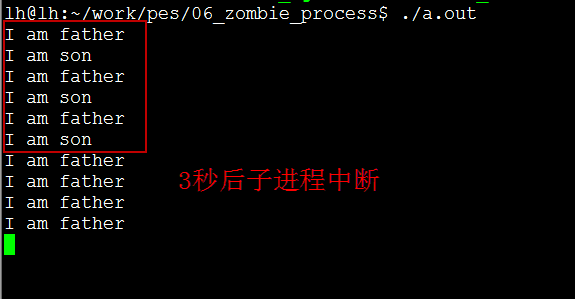
        }

    }

    return 0;

}

运行结果：



# 等待信号

#include <unistd.h>

int pause(void);

功能：

等待信号的到来（此函数会阻塞）。

参数：

无。

返回值：

直到捕获到信号才返回 -1，且 errno 被设置成 EINTR。

测试代码如下：

#include <unistd.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

    printf("in pause function\n");

    pause();

    return 0;

}

没有产生信号前，进程一直阻塞在 pause() 不会往下执行，假如，我们按“Ctrl+c”，pause() 会捕获到此信号，中断当前进程

# 处理信号

#include <signal.h>

// 回调函数的声明

typedef void (\*sighandler\_t)(int);

// 注册信号处理函数

sighandler\_t signal(int signum, sighandler\_t handler);

注：SIGKILL 和 SIGSTOP （终止进程）不能更改信号的处理方式

功能：  
 注册信号处理函数（不可用于 SIGKILL、SIGSTOP 信号）  
参数：  
 signum：信号的编号，这里可以填数字编号，也可以填信号的宏定义  
 handler: 取值有 3 种情况：  
 SIG\_IGN：忽略该信号  
 SIG\_DFL：执行系统默认动作  
 信号处理函数名：自定义信号处理函数  
返回值：  
 成功：返回此信号上一次注册的信号处理函数的地址。  
 失败：返回 SIG\_ERR

示例

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

// 信号处理函数

void signal\_handler(int signo)

{

    if(signo == SIGINT){

        printf("recv SIGINT\n");

    }else if(signo == SIGQUIT){

        printf("recv SIGQUIT\n");

    }

}

int main(int argc, char \*argv[])

{

    printf("wait for SIGINT OR SIGQUIT\n");

    /\* SIGINT: Ctrl+c ; SIGQUIT: Ctrl+\ \*/

    // 信号注册函数

    signal(SIGINT, signal\_handler);

    signal(SIGQUIT, signal\_handler);

    // 等待信号

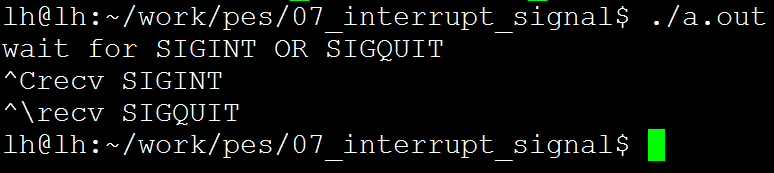
    pause();

    pause();

    return 0;

}

运行结果



# 信号集与信号阻塞集

## 信号集（信号的集合）

为了方便对多个信号进行处理，在 Linux 系统中引入了信号集

sigset\_t代表一个信号集  
信号集操作：

#include <signal.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

    sigset\_t set;   // 定义一个信号集

    sigemptyset(&set); // 清空信号集的内容

    sigaddset(&set, SIGINT); // 把 SIGINT 添加到信号集 set

    sigaddset(&set, SIGQUIT);// 把 SIGQUIT 添加到信号集 set

    // 判断 SIGINT 是否在信号集 set 里，在返回 1， 不在返回 0

    if(sigismember(&set, SIGINT) == 1){

        printf("SIGINT is a member of set \nret = %d\n", ret);

    }

    sigdelset(&set, SIGQUIT); // 把 SIGQUIT 从信号集 set 移除

    return 0;

}

## 信号阻塞集(屏蔽集、掩码)

进程收到属于阻塞集的信号时会阻塞不执行，直到从阻塞集移除该信号  
通过 sigprocmask() 修改当前进程的信号阻塞集。

#include <signal.h>

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

功能：

阻塞信号

参数：

how: 信号阻塞集合的修改方法，有 3 种情况：

SIG\_BLOCK：向阻塞集添加set指定的信号集

SIG\_UNBLOCK：阻塞集中删除set指定的信号集

SIG\_SETMASK：用set替换旧的阻塞集

set: 要操作的信号集地址。

oldset: 保存原先信号阻塞集地址  
返回值：

成功：0，

失败：-1，失败时错误代码只可能是 EINVAL，表示参数 how 不合法。

示例代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

int main(int argc, char \*argv[])

{

    sigset\_t set; // 信号集合

    int i = 0;

    sigemptyset(&set); // 清空信号集合

    sigaddset(&set, SIGINT); // SIGINT 加入 set 集合

    while(1)

    {

        // set 集合加入阻塞集，在没有移除前，SIGINT 会被阻塞

        sigprocmask(SIG\_BLOCK, &set, NULL);

        for(i=0; i<5; i++)

        {

            printf("SIGINT signal is blocked\n");

            sleep(1);

        }

        // set 集合从阻塞集中移除

        // 如果 SIGINT 信号在阻塞时发生，SIGINT 信号立马生效，中断当前进程

        sigprocmask(SIG\_UNBLOCK, &set, NULL);

        for(i=0; i<5; i++)

        {

            printf("没有收到 SIGINT 信号\n");

            sleep(1);

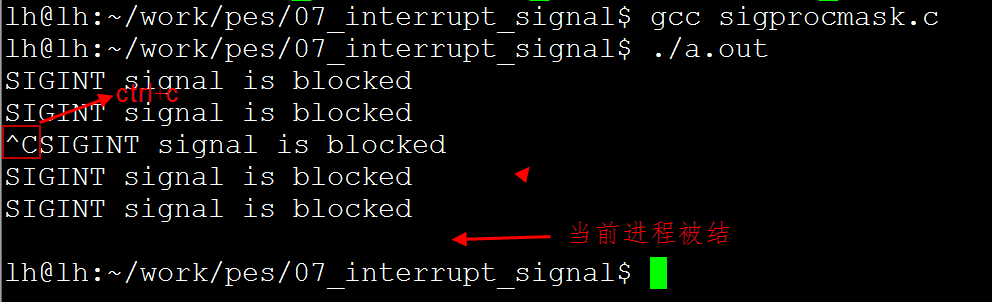
        }

    }

    return 0;

}

运行结果：



## 可携带数据的信号操作

所需头文件：

#include <signal.h>

**发送信号**

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

功能：

给指定进程发送信号。

参数：

pid: 进程号。

sig: 信号的编号，这里可以填数字编号，也可以填信号的宏定义

value:  通过信号传递的参数。

union sigval 类型如下：

union sigval  {

    int   sival\_int;

    void \*sival\_ptr;

};

注：sival\_ptr不能用于跨域不同的程序传值，sigqueue只会将sival\_ptr指向的地址拷贝给另一个进程，如，发送信号时sival\_ptr指向0x123，那么接收信号的进程收到的sival\_ptr也是0x123，如果接收信号的进程0x123地址没有值，那么就会出现段错误

返回值：

成功：0

失败：-1

**指定信号处理函数**

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

功能：

检查或修改指定信号的处理方式（或同时执行这两种操作）。

参数：

signum：要操作的信号。

act：要设置的对信号的新处理方式。

oldact：

原来对信号的处理方式。

如果 oldact 指针非空，则系统将此前指定信号的处理方式（设置）存入 oldact。

返回值：

成功：0

失败：-1

**信号处理方式结构体**

struct sigaction  {

    /\* 不接收数据的处理函数 \*/

    void (\*sa\_handler)(int signum) ;

    /\* 接收数据的处理函数 \*/

    void (\*sa\_sigaction)(int signum, siginfo\_t \*info, void \*context);

    /\*信号阻塞集\*/

    sigset\_t sa\_mask;

    /\*信号处理的方式\*/

    int sa\_flags;

};

sa\_handler、sa\_sigaction：

信号处理函数指针，二选一赋值，其取值如下：

SIG\_IGN：忽略该信号

SIG\_DFL：执行系统默认动作

处理函数名：自定义信号处理函数

sa\_mask：信号阻塞集

sa\_flags：用于指定信号处理的行为，“按位或”组合：

SA\_NOCLDSTOP：父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号

SA\_NOCLDWAIT：父进程在它的子进程退出时不会收到 SIGCHLD 信号

SA\_NODEFER：使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号

SA\_RESETHAND：信号处理之后重新设置为默认的处理方式

SA\_SIGINFO：使用 sa\_sigaction 成员而不是 sa\_handler 作为信号处理函数

**信号处理函数**

void (\*sa\_sigaction)( int signum, siginfo\_t \*info, void \*context );

参数：

signum：信号的编号。  
 info：

记录信号发送进程信息的结构体，进程信息结构体路/usr/include/i386-linux-gnu/bits/siginfo.

context：

可以赋给指向 ucontext\_t 类型的一个对象的指针，以引用在传递信号时被中断的接收进程或线程的上下文

**发送信号示例**

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <unistd.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*功能：     发 SIGINT 信号及信号携带的值给指定的进程

\*参数：     argv[1]：进程号

          argv[2]：待发送的值（默认为100）

\*返回值：   0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \*argv[])

{

    if(argc >= 2)

    {

        pid\_t pid,pid\_self;

        union sigval tmp;

        pid = atoi(argv[1]); // 进程号

        if( argc >= 3 )

        {

            tmp.sival\_int = atoi(argv[2]);

        }

        else

        {

            tmp.sival\_int = 100;

        }

        // 给进程 pid，发送 SIGINT 信号，并把 tmp 传递过去

        sigqueue(pid, SIGINT, tmp);

        pid\_self = getpid(); // 进程号

        printf("pid = %d, pid\_self = %d\n", pid, pid\_self);

    }

    return 0;

}

接收信号示例代码如下：

#include <signal.h>

#include <stdio.h>

// 信号处理回电函数

void signal\_handler(int signum, siginfo\_t \*info, void \*ptr)

{

    printf("signum = %d\n", signum); // 信号编号

    printf("info->si\_pid = %d\n", info->si\_pid); // 对方的进程号

    printf("info->si\_sigval = %d\n", info->si\_value.sival\_int); // 对方传递过来的信息

}

int main(int argc, char \*argv[])

{

    struct sigaction act, oact;

    act.sa\_sigaction = signal\_handler; //指定信号处理回调函数

    sigemptyset(&act.sa\_mask); // 阻塞集为空

    act.sa\_flags = SA\_SIGINFO; // 指定调用 signal\_handler

    // 注册信号 SIGINT

    sigaction(SIGINT, &act, &oact);

    while(1)

    {

        printf("pid is %d\n", getpid()); // 进程号

        pause(); // 捕获信号，此函数会阻塞

    }

    return 0;

}

两个终端分别编译代码，一个进程接收，一个进程发送，运行结果如下：

