LINUX进程通信

- ◆ 经典通信方法（续）
- ◆ SYSTEM V IPC

（二）信号机制（SIGNAL）

为什么需要SIGNAL机制?

程序的运行模式

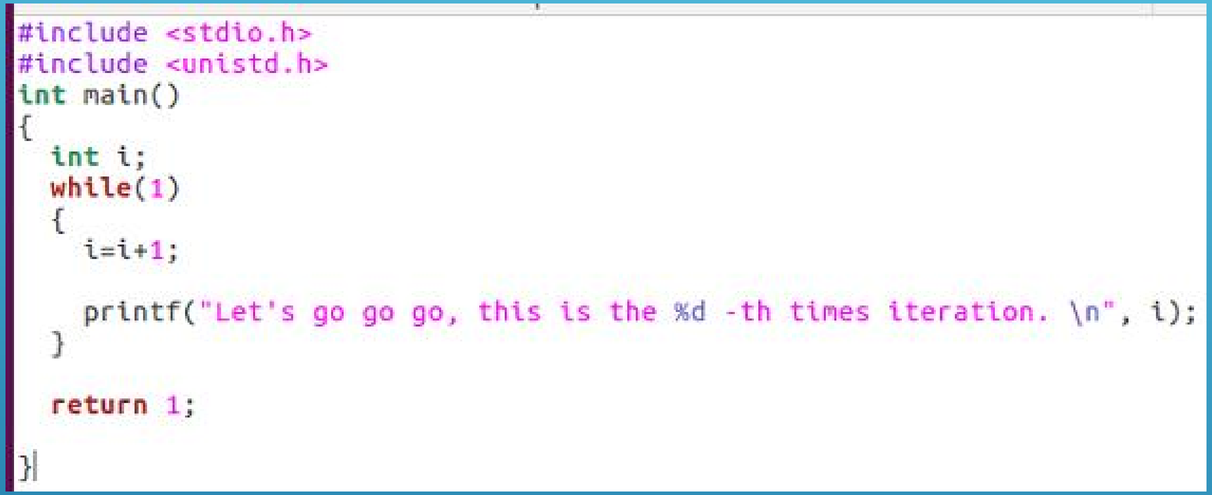
- 前台进程
  - 显示占据终端、人机交互、人工控制、运行结束或人为终止
- 后台进程
  - 后台服务、系统控制、无需交互、可以没有界面

（二）信号机制（SIGNAL）

守护进程

- 一种特殊的后台进程，不能直接和用户交互
- 独立于控制终端执行某些周期性的系统服务或等待某些事件
    - FTP服务：ftpd
    - apache超文本传输：httpd
    - 网络守护进程：xinetd
    - 作业规划守护进程：crond

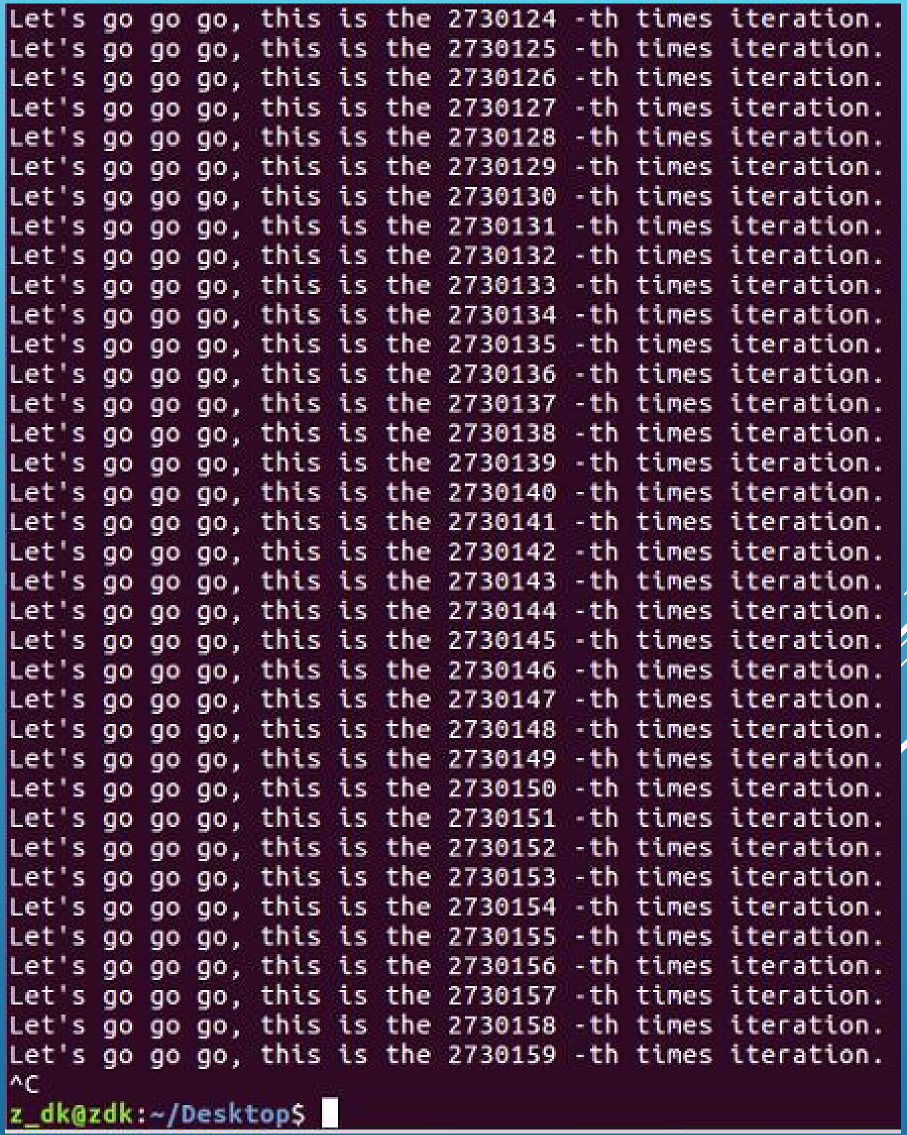
进程不同运行模式的控制



```c
#include <stdio.h>
#include <unistd.h>
int main()
{
  int i;
  while(1)
  {
    i=i+1;

    printf("Let's go go go, this is the %d -th times iteration. \n", i);
  }

  return 1;

}
```

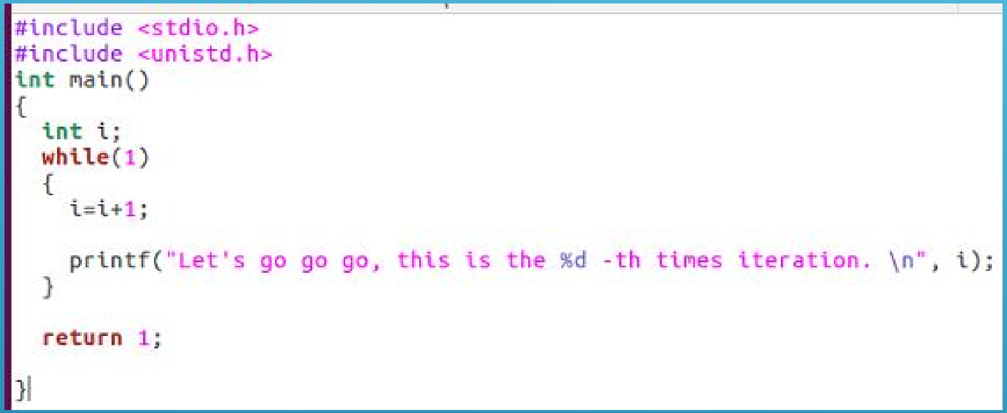
常规前台运行
./loop

终端键入Ctrl+C　正常结束



```
Let's go go go, this is the 2730124 -th times iteration.
Let's go go go, this is the 2730125 -th times iteration.
Let's go go go, this is the 2730126 -th times iteration.
Let's go go go, this is the 2730127 -th times iteration.
Let's go go go, this is the 2730128 -th times iteration.
Let's go go go, this is the 2730129 -th times iteration.
Let's go go go, this is the 2730130 -th times iteration.
Let's go go go, this is the 2730131 -th times iteration.
Let's go go go, this is the 2730132 -th times iteration.
Let's go go go, this is the 2730133 -th times iteration.
Let's go go go, this is the 2730134 -th times iteration.
Let's go go go, this is the 2730135 -th times iteration.
Let's go go go, this is the 2730136 -th times iteration.
Let's go go go, this is the 2730137 -th times iteration.
Let's go go go, this is the 2730138 -th times iteration.
Let's go go go, this is the 2730139 -th times iteration.
Let's go go go, this is the 2730140 -th times iteration.
Let's go go go, this is the 2730141 -th times iteration.
Let's go go go, this is the 2730142 -th times iteration.
Let's go go go, this is the 2730143 -th times iteration.
Let's go go go, this is the 2730144 -th times iteration.
Let's go go go, this is the 2730145 -th times iteration.
Let's go go go, this is the 2730146 -th times iteration.
Let's go go go, this is the 2730147 -th times iteration.
Let's go go go, this is the 2730148 -th times iteration.
Let's go go go, this is the 2730149 -th times iteration.
Let's go go go, this is the 2730150 -th times iteration.
Let's go go go, this is the 2730151 -th times iteration.
Let's go go go, this is the 2730152 -th times iteration.
Let's go go go, this is the 2730153 -th times iteration.
Let's go go go, this is the 2730154 -th times iteration.
Let's go go go, this is the 2730155 -th times iteration.
Let's go go go, this is the 2730156 -th times iteration.
Let's go go go, this is the 2730157 -th times iteration.
Let's go go go, this is the 2730158 -th times iteration.
Let's go go go, this is the 2730159 -th times iteration.
^C
z_dk@zdk:~/Desktop$
```

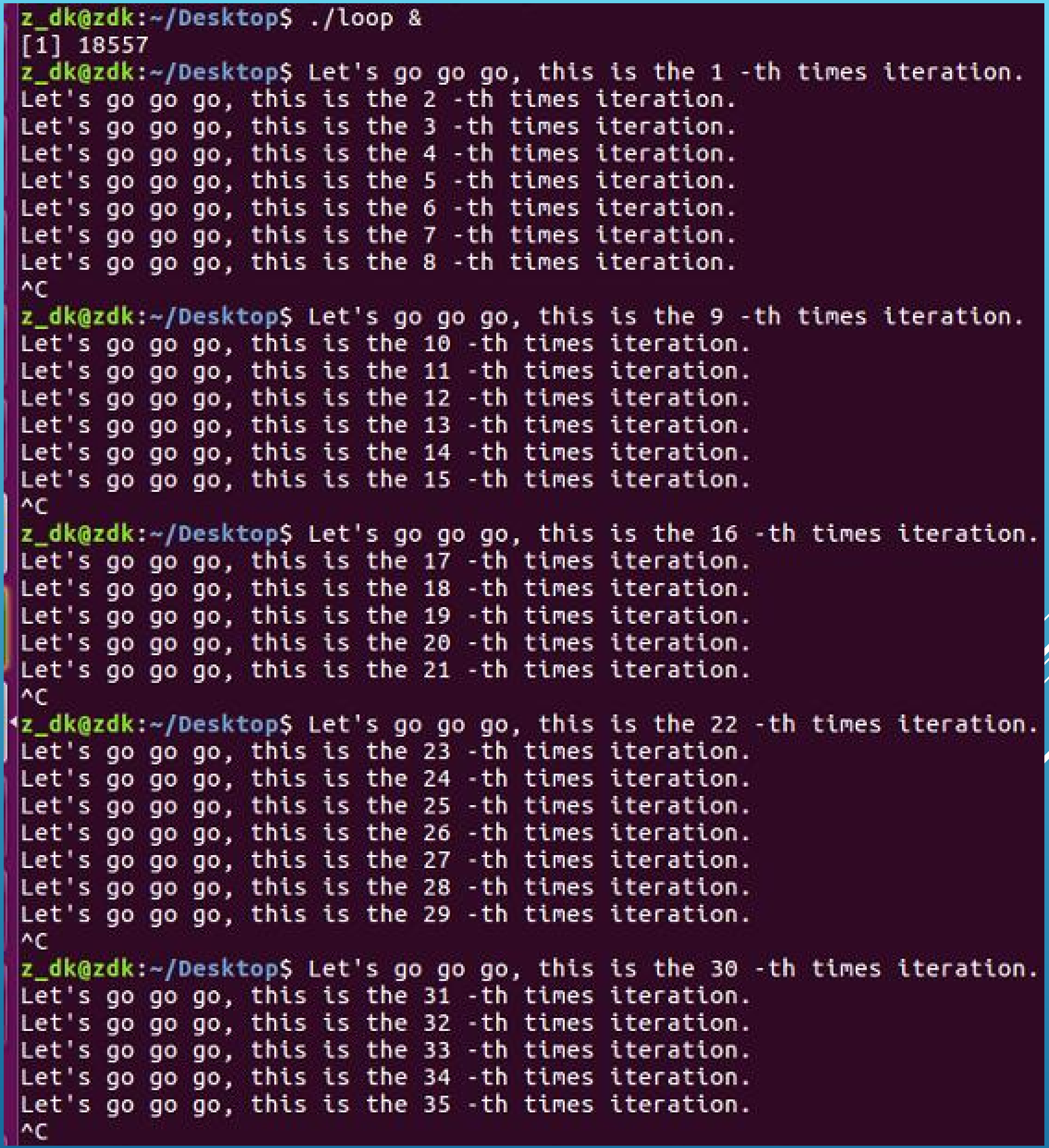
进程不同运行模式的控制



```c
#include <stdio.h>
#include <unistd.h>
int main()
{
  int i;
  while(1)
  {
    i=i+1;

    printf("Let's go go go, this is the %d -th times iteration. \n", i);
  }

  return 1;

}
```

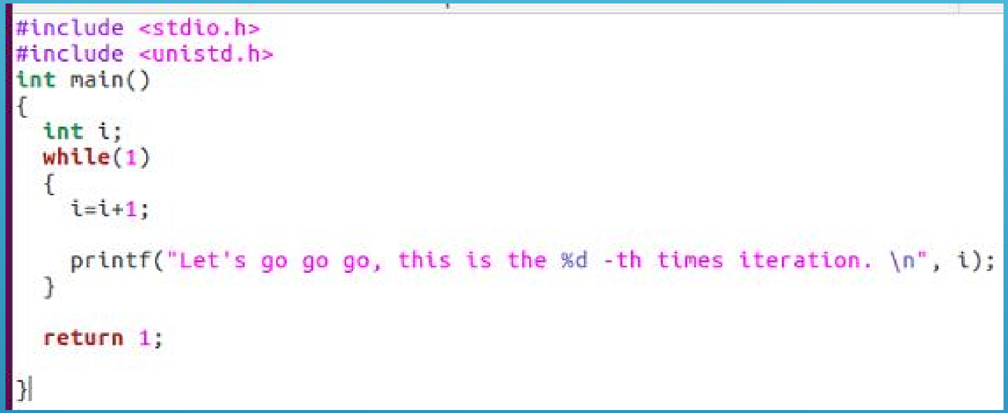
后台运行
./loop &

终端键入Ctrl+C 程序无响应

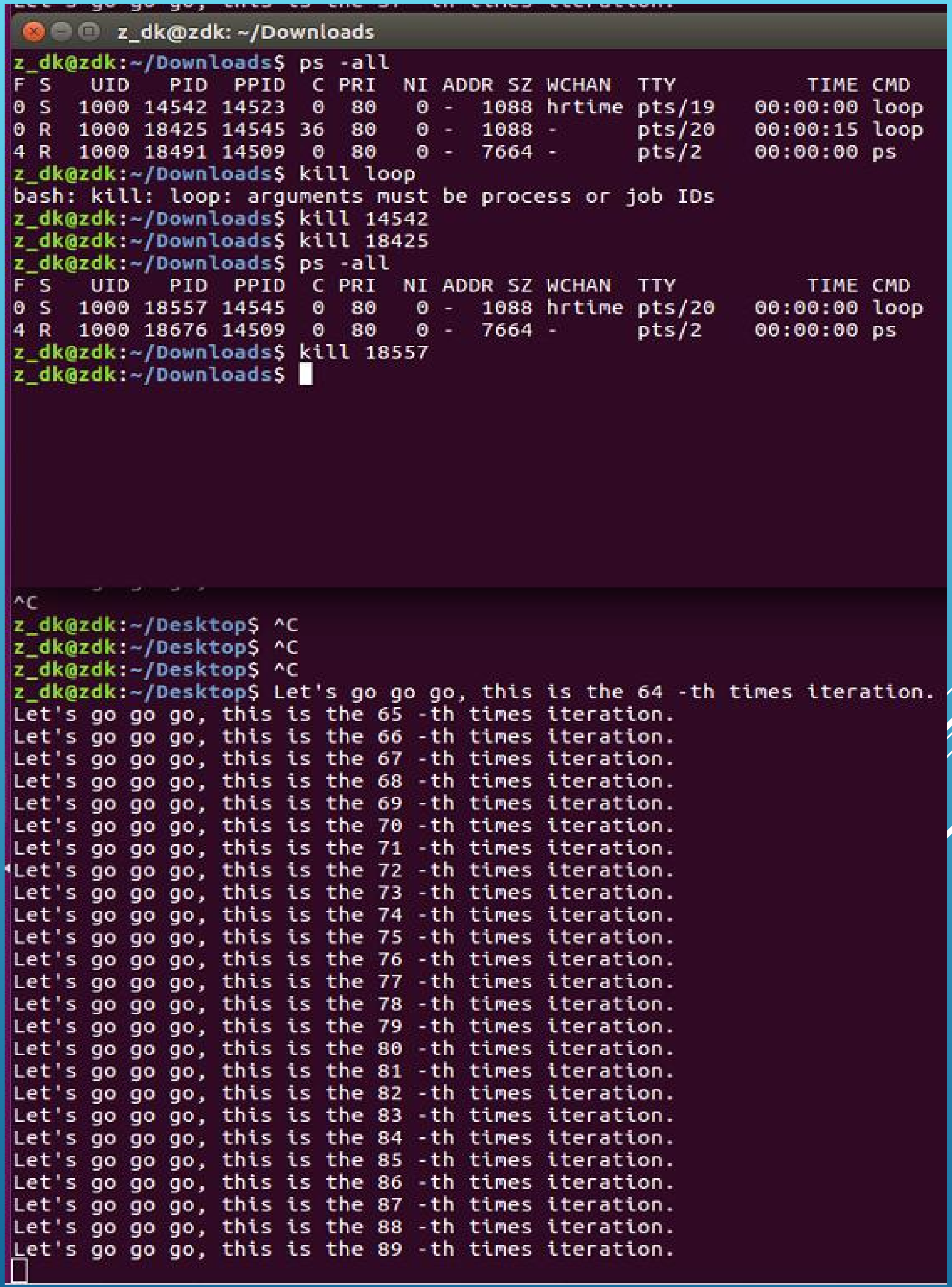


```
z_dk@zdk:~/Desktop$ ./loop &
[1] 18557
z_dk@zdk:~/Desktop$ Let's go go go, this is the 1 -th times iteration.
Let's go go go, this is the 2 -th times iteration.
Let's go go go, this is the 3 -th times iteration.
Let's go go go, this is the 4 -th times iteration.
Let's go go go, this is the 5 -th times iteration.
Let's go go go, this is the 6 -th times iteration.
Let's go go go, this is the 7 -th times iteration.
Let's go go go, this is the 8 -th times iteration.
^C
z_dk@zdk:~/Desktop$ Let's go go go, this is the 9 -th times iteration.
Let's go go go, this is the 10 -th times iteration.
Let's go go go, this is the 11 -th times iteration.
Let's go go go, this is the 12 -th times iteration.
Let's go go go, this is the 13 -th times iteration.
Let's go go go, this is the 14 -th times iteration.
Let's go go go, this is the 15 -th times iteration.
^C
z_dk@zdk:~/Desktop$ Let's go go go, this is the 16 -th times iteration.
Let's go go go, this is the 17 -th times iteration.
Let's go go go, this is the 18 -th times iteration.
Let's go go go, this is the 19 -th times iteration.
Let's go go go, this is the 20 -th times iteration.
Let's go go go, this is the 21 -th times iteration.
^C
z_dk@zdk:~/Desktop$ Let's go go go, this is the 22 -th times iteration.
Let's go go go, this is the 23 -th times iteration.
Let's go go go, this is the 24 -th times iteration.
Let's go go go, this is the 25 -th times iteration.
Let's go go go, this is the 26 -th times iteration.
Let's go go go, this is the 27 -th times iteration.
Let's go go go, this is the 28 -th times iteration.
Let's go go go, this is the 29 -th times iteration.
^C
z_dk@zdk:~/Desktop$ Let's go go go, this is the 30 -th times iteration.
Let's go go go, this is the 31 -th times iteration.
Let's go go go, this is the 32 -th times iteration.
Let's go go go, this is the 33 -th times iteration.
Let's go go go, this is the 34 -th times iteration.
Let's go go go, this is the 35 -th times iteration.
^C
```

进程不同运行模式的控制



```c
#include <stdio.h>
#include <unistd.h>
int main()
{
  int i;
  while(1)
  {
    i=i+1;

    printf("Let's go go go, this is the %d -th times iteration. \n", i);
  }

  return 1;

}
```



```
z_dk@zdk: ~/Downloads
z_dk@zdk:~/Downloads$ ps -all
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 14542 14523  0  80   0 -  1088 hrtime pts/19   00:00:00 loop
0 R  1000 18425 14545 36  80   0 -  1088 -      pts/20   00:00:15 loop
4 R  1000 18491 14509  0  80   0 -  7664 -      pts/2    00:00:00 ps
z_dk@zdk:~/Downloads$ kill loop
bash: kill: loop: arguments must be process or job IDs
z_dk@zdk:~/Downloads$ kill 14542
z_dk@zdk:~/Downloads$ kill 18425
z_dk@zdk:~/Downloads$ ps -all
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 18557 14545  0  80   0 -  1088 hrtime pts/20   00:00:00 loop
4 R  1000 18676 14509  0  80   0 -  7664 -      pts/2    00:00:00 ps
z_dk@zdk:~/Downloads$ kill 18557
z_dk@zdk:~/Downloads$ 
```

```
^C
z_dk@zdk:~/Desktop$ ^C
z_dk@zdk:~/Desktop$ ^C
z_dk@zdk:~/Desktop$ ^C
z_dk@zdk:~/Desktop$ Let's go go go, this is the 64 -th times iteration.
Let's go go go, this is the 65 -th times iteration.
Let's go go go, this is the 66 -th times iteration.
Let's go go go, this is the 67 -th times iteration.
Let's go go go, this is the 68 -th times iteration.
Let's go go go, this is the 69 -th times iteration.
Let's go go go, this is the 70 -th times iteration.
Let's go go go, this is the 71 -th times iteration.
Let's go go go, this is the 72 -th times iteration.
Let's go go go, this is the 73 -th times iteration.
Let's go go go, this is the 74 -th times iteration.
Let's go go go, this is the 75 -th times iteration.
Let's go go go, this is the 76 -th times iteration.
Let's go go go, this is the 77 -th times iteration.
Let's go go go, this is the 78 -th times iteration.
Let's go go go, this is the 79 -th times iteration.
Let's go go go, this is the 80 -th times iteration.
Let's go go go, this is the 81 -th times iteration.
Let's go go go, this is the 82 -th times iteration.
Let's go go go, this is the 83 -th times iteration.
Let's go go go, this is the 84 -th times iteration.
Let's go go go, this is the 85 -th times iteration.
Let's go go go, this is the 86 -th times iteration.
Let's go go go, this is the 87 -th times iteration.
Let's go go go, this is the 88 -th times iteration.
Let's go go go, this is the 89 -th times iteration.
```

后台运行
./loop　&

本终端被后台运行进程独占
新开终端，查询该后台运行进程ID
通过kill发送信号终止该进程
成功终止

（二）信号机制（SIGNAL）

信号：（signal，又称为软中断信号）用来通知进程发生了异步事件（不可预知），在软件层次上是对中断机制的一种模拟。

信号来源：

1）硬件

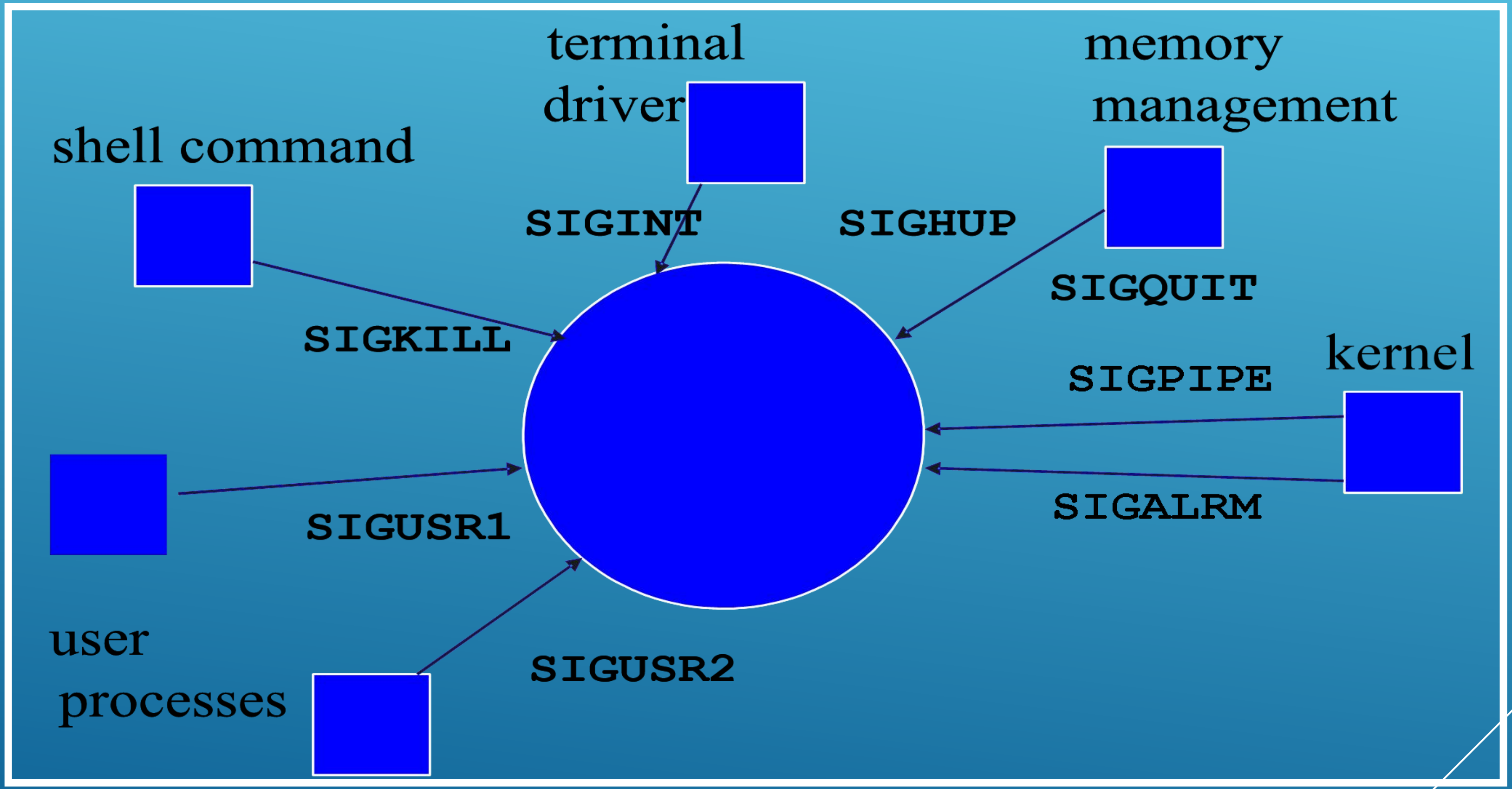
① 用户按键，如**Ctrl+C**

② 硬件异常，如零除，内存引用错误（内核通知该进程）

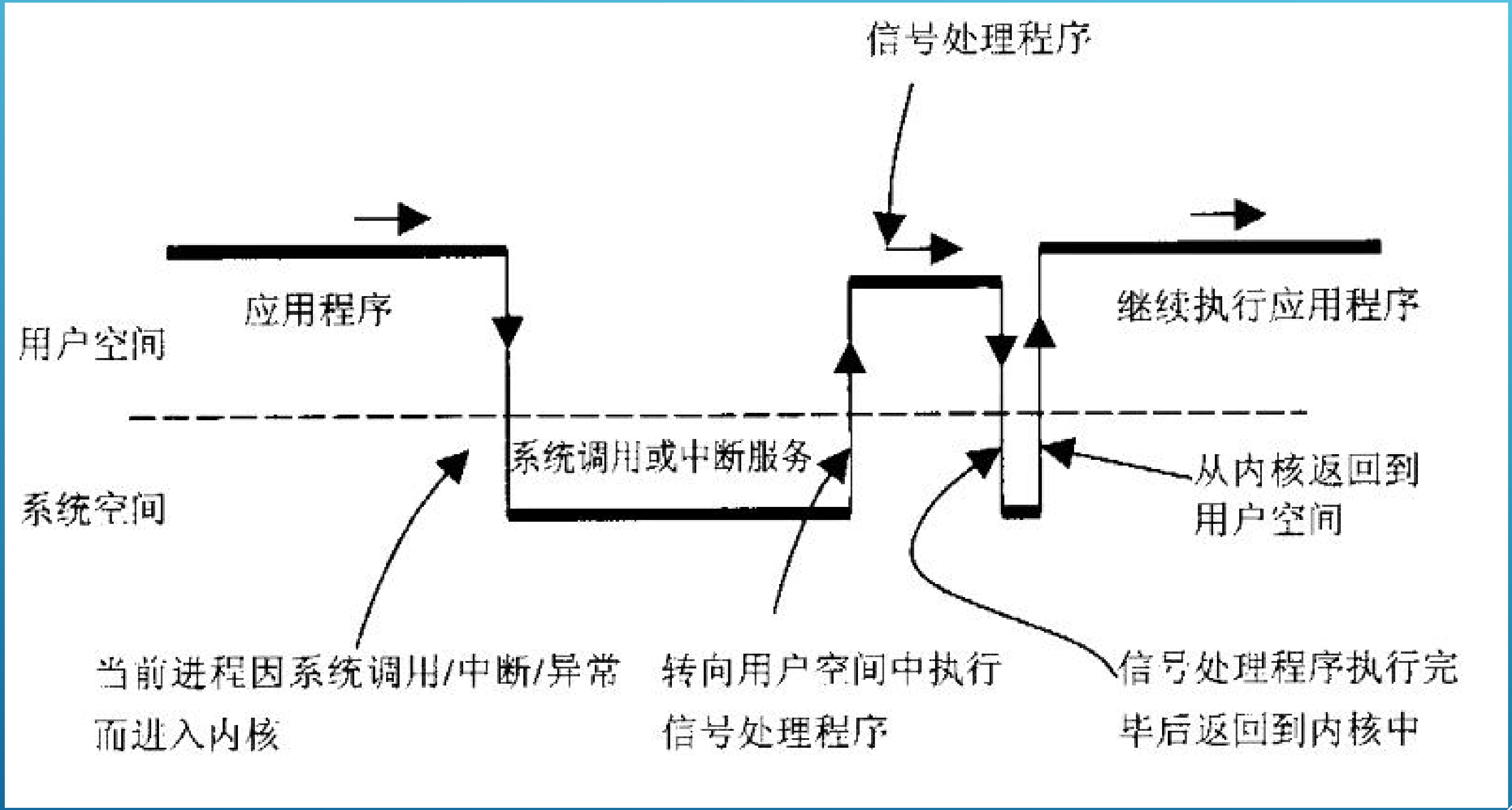
2）软件

通过系统调用kill，raise，alarm等产生

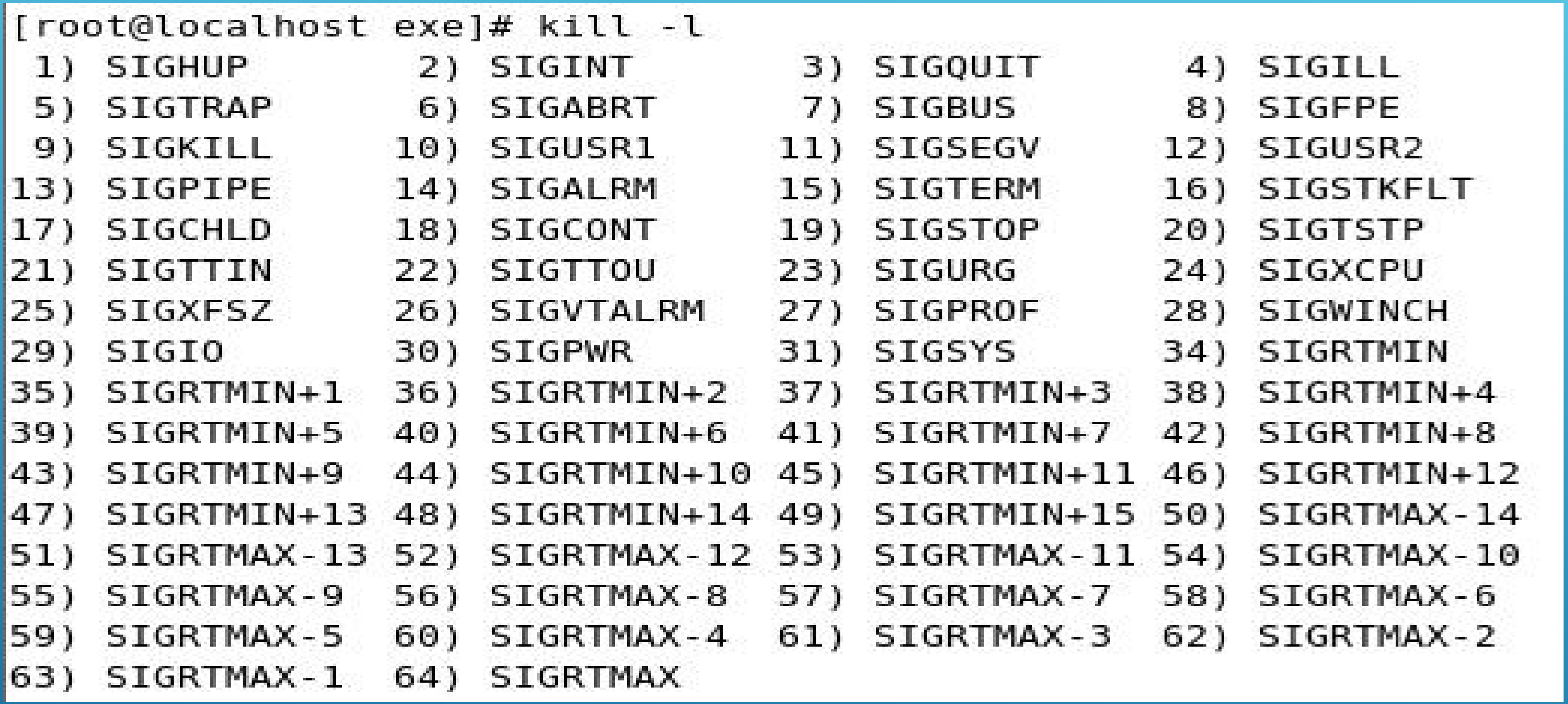
信号的来源


terminal driver
memory management
shell command
SIGINT
SIGHUP
SIGQUIT
SIGKILL
kernel
SIGPIPE
SIGALRM
SIGUSR1
user processes
SIGUSR2

信号检测与处理流程


信号处理程序
应用程序
用户空间
继续执行应用程序
系统调用或中断服务
系统空间
从内核返回到用户空间
当前进程因系统调用/中断/异常而进入内核
转向用户空间中执行信号处理程序
信号处理程序执行完毕后返回到内核中

LINUX信号定义



```
[root@localhost exe]# 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
```

1-31： 非实时信号（不可靠）
34-64：实时信号(可靠)

LINUX常用信号

SIGINT: 按键ˆC产生，默认操作终止当前前台进程；

SIGHUP： 从终端上发出的挂起信号；

SIGQUIT: 按键ˆ \ 产生，类似于程序错误信号

SIGFPE： 浮点异常信号（例如浮点运算溢出）；

SIGKILL：该信号结束接收信号的进程；

SIGALRM：进程的定时器到期时，发送该信号；

SIGTERM：中止信号，程序自动退出，kill的默认发出信号；

SIGCHLD：标识子进程停止或结束的信号；

SIGSTOP：来自键盘（Ctrl-Z）或调试程序的暂停执行信号

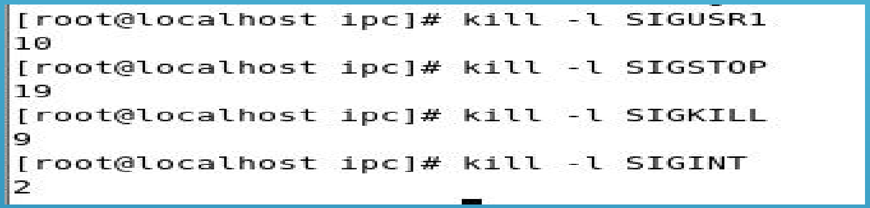
信号控制进程命令

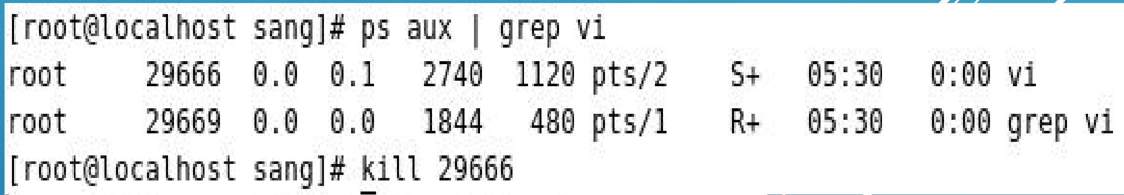
Kill　[参数]　[进程ID]

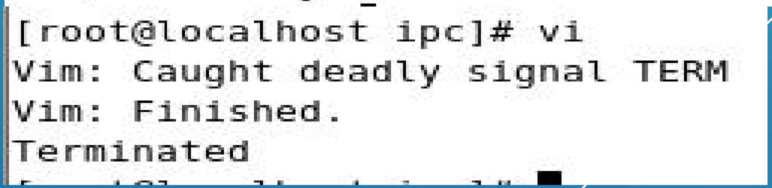
- Kill －l　信号名：返回信号的值（编号）
- Kill －l：　列出系统定义的所有信号
- Ps查找到进程，并且kill掉
- Kill　-s　发送指定信号

pkill　进程名

杀掉进程名对应的进程


```
[root@localhost ipc]# kill -l SIGUSR1
10
[root@localhost ipc]# kill -l SIGSTOP
19
[root@localhost ipc]# kill -l SIGKILL
9
[root@localhost ipc]# kill -l SIGINT
2
```


```
[root@localhost sang]# ps aux | grep vi
root     29666  0.0  0.1   2740  1120 pts/2    S+   05:30   0:00 vi
root     29669  0.0  0.0   1844   480 pts/1    R+   05:30   0:00 grep vi
[root@localhost sang]# kill 29666
```


```
[root@localhost ipc]# vi
Vim: Caught deadly signal TERM
Vim: Finished.
Terminated
```

信号相关基本概念

生成（generate）：引发信号的事件发生

送达（deliver）： 信号被送达到进程

未决（pending）： 信号产生（generate），但尚未送达(deliver)

阻塞（block）： 信号不会Deliver，保持在未决状态；解除block，才会送达

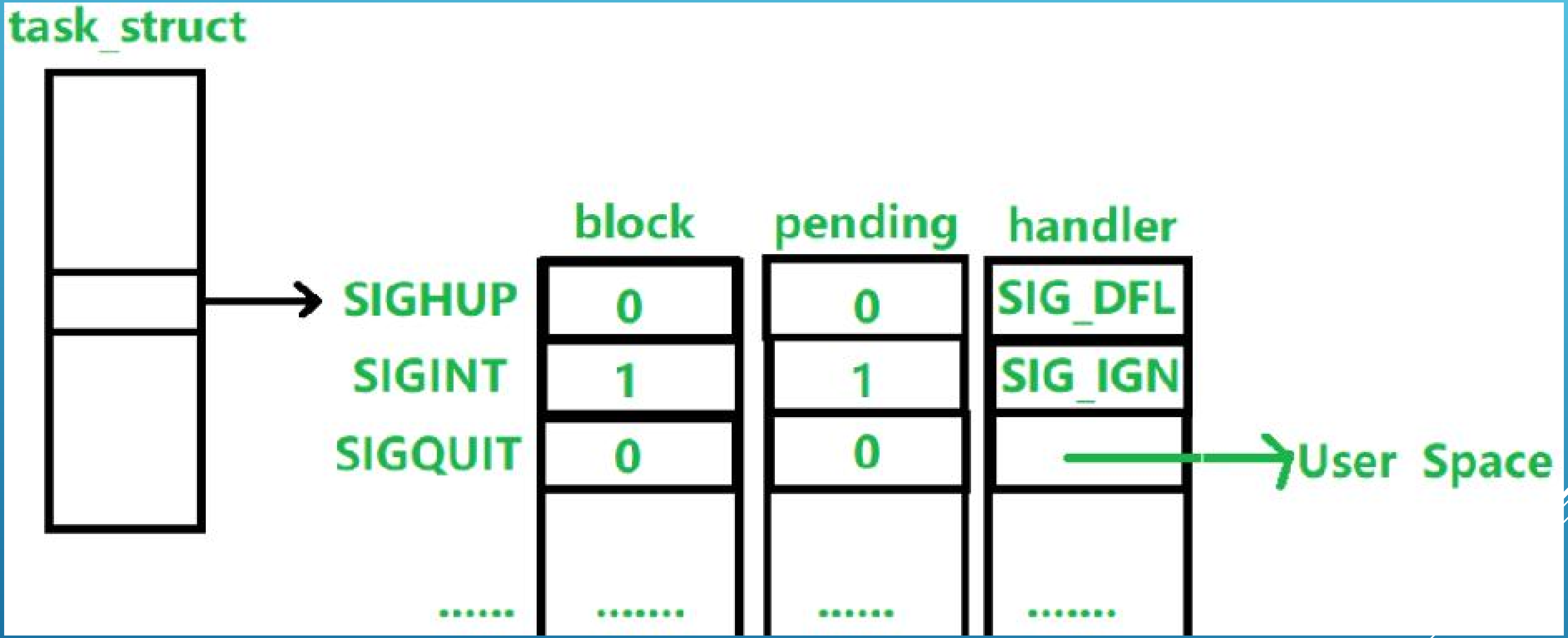
捕获（catch）： Deliver时进程执行了signal handler=进程捕获到该信号

忽略（SIG_IGN）： 信号Deliver，但不处理

可靠/不可靠：不可靠，则同种信号将覆盖之前未处理的该类信号，即在接收进程中只能有一个该类未决成员；可靠信号则会存放在队列中，即有多个未决成员。

2. 信号的表示

PCB(TASK_STRUCT)



3.　信号产生

- ❑ **内核（隐式）产生（检测到系统事件）**
  - ✓ 硬件异常，如零除，内存访问失败等由硬件检测并通知内核，内核根据情况产生合适的信号发送给进程
- ❑ **用户通过终端按键（显式）产生信号**
  - ✓ 如按键盘Ctrl+C组合键产生信号(SIGINT)
  - ✓ 如按键盘Ctrl+Z组合键产生信号(SIGQUIT)
- ❑ **软件的原因产生信号（显式）**
  - ✓ 使用系统函数：　kill,raise,alarm,abort函数
  - ✓ 命令发出：kill命令

4. 信号处理方式

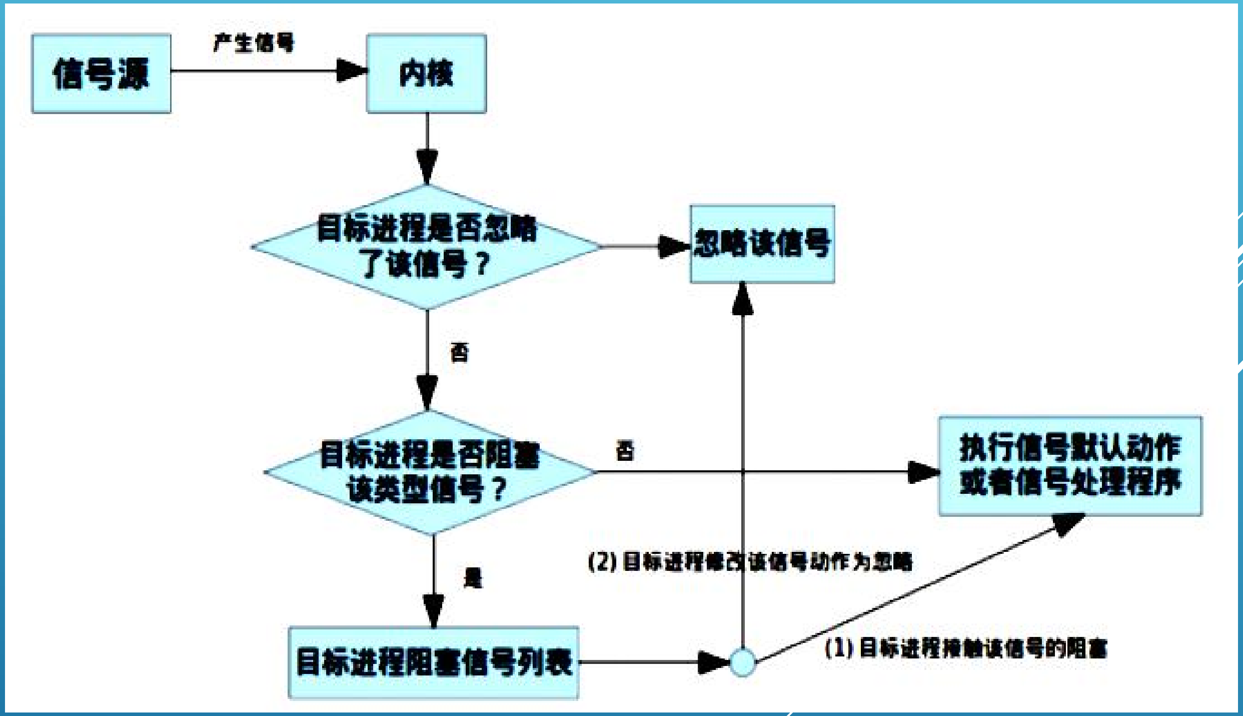
- 进程收到信号后可能的处理方式：

1） **忽略**信号（除SIGKILL和SIGSTOP外）

2） **阻塞**信号

3） **进程处理**该信号（捕捉信号，通知内核调用户函数对该事件做特定处理）

4） 系统进行**默认处理**（多数信号的默认动作是终止该进程）


信号源
产生信号
内核
目标进程是否忽略了该信号？
忽略该信号
否
目标进程是否阻塞该类型信号？
否
执行信号默认动作或者信号处理程序
是
(2) 目标进程修改该信号动作为忽略
目标进程阻塞信号列表
(1) 目标进程解除该信号的阻塞

4. 信号处理方式

阻塞信号

每个进程都有一个信号掩码(signal mask),它设置了当前要阻塞传递给该进程的信号集;

每种信号在掩码中都对应了一位 。

sigprocmask 函数：可以检测或者设置进程的信号屏蔽字

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);
```

返回值:若成功则返回0,若出错则返回-1

5. 信号的安装与处理

- ✓ 如果对某进程不进行任何信号的控制，收到信号将安装缺省的方式进行处理；如果希望进程对收到信号按照某种自定义的方式处理，则必须对进程进行信号的安装。
- ✓ 安装的实质是指定收到某种信号后进行哪些具体的操作，即调用哪个函数，这个函数是用户可以自定义的。
- ✓ 信号安装（系统调用）：
  -SIGNAL函数
  -SIGACTION函数

- 函数调用形式：

```
#include<signal.h>
int kill(pid_t pid,int sig);
int raise(int sig);
unisigned int alarm(unsigned int seconds);
```

- 作用：
  - kill系统调用负责向进程发送信号sig（对应的进程PCB相应队列会增加节点）
  - raise系统调用向自己发送一个sig信号；
  - alarm可以在seconds秒后向自己发送一个SIGALRM信号。
- 参数：

  如果pid是正数,那么信号sig被发送到进程pid.

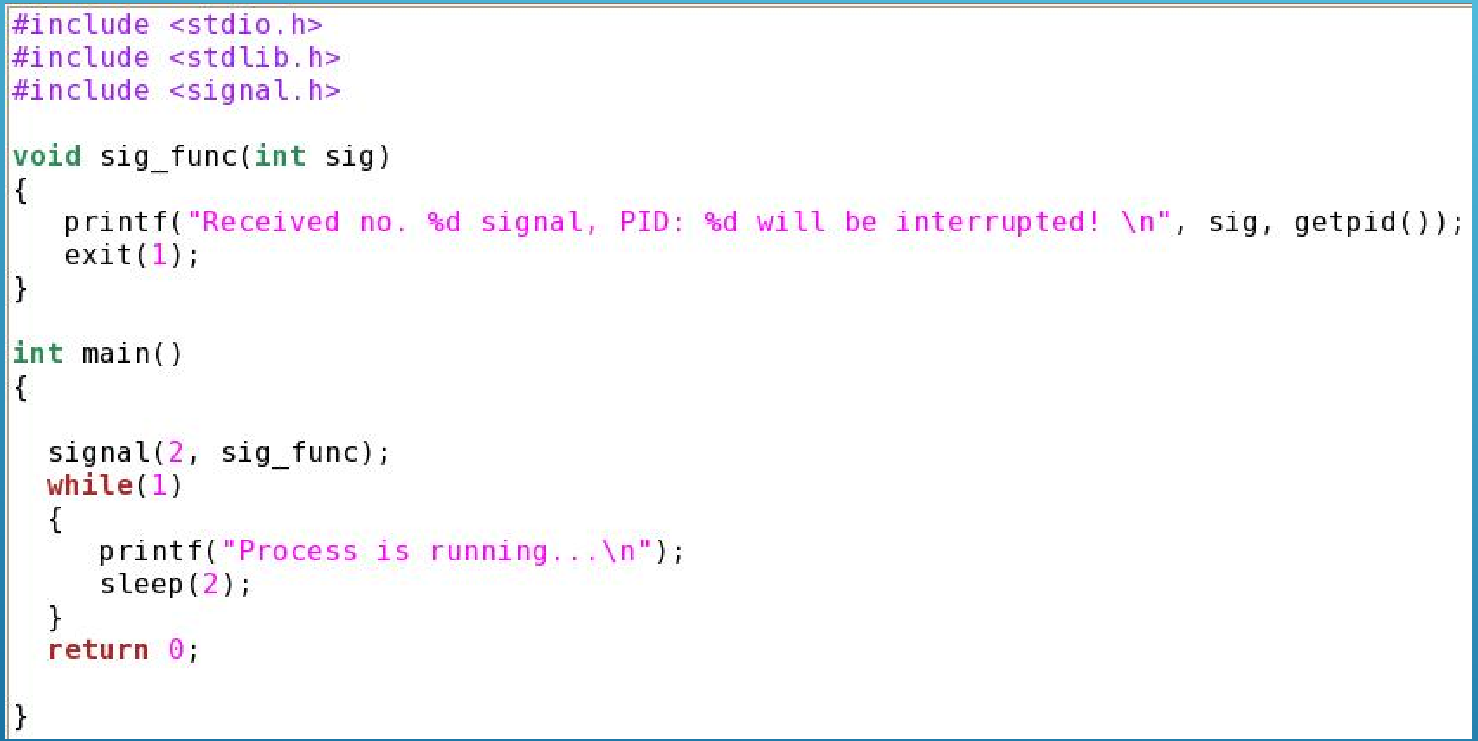
  如果pid等于0,那么信号sig被发送到和pid进程在同一个进程组的进程

  如果pid等于-1,那么信号发给所有的进程表中的进程,除了最大的进程号

signal()信号处理函数

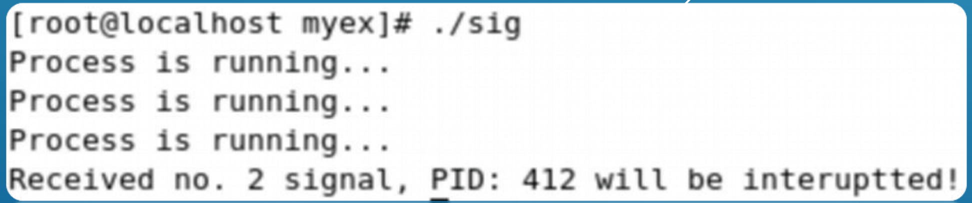
头文件	#include <signal.h>
原型	void (*signal(int signum,void(*handler)(int)))(int)
参数	signum：指定信号
	handler：指定信号处理的方式 SIG_IGN：忽略该信号 SIG_DFL：采用系统默认的方式处理信号 自定义的信号处理函数指针
返回值	成功：以前的信号处理配置
	出错：－1

例子（用signal函数安装信号处理函数）



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

void sig_func(int sig)
{
   printf("Received no. %d signal, PID: %d will be interrupted! \n", sig, getpid());
   exit(1);
}

int main()
{

  signal(2, sig_func);
  while(1)
  {
    printf("Process is running...\n");
    sleep(2);
  }
  return 0;

}
```

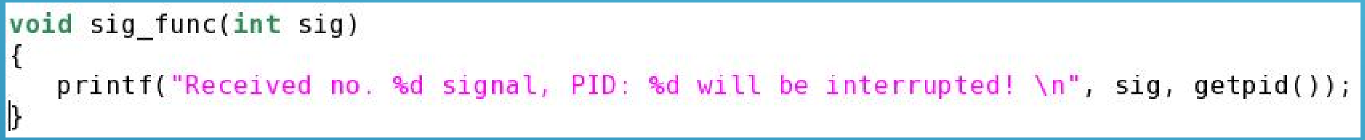
发送信号：
1）在当前程序运行终端键入“Ctrl+C”
2）在另外一个不同的终端“kill - s 2 进程号”



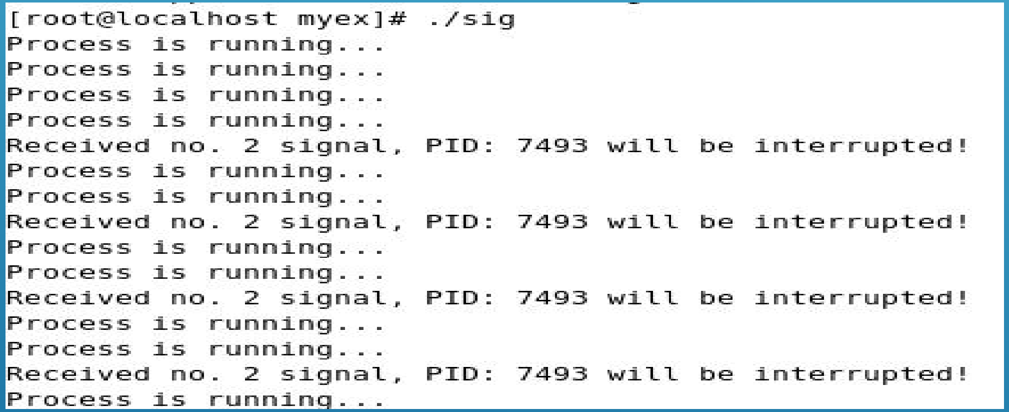
```
[root@localhost myex]# ./sig
Process is running...
Process is running...
Process is running...
Received no. 2 signal, PID: 412 will be interuptted!
```

例子（用signal函数安装信号处理函数）

注意1：如果信号处理函数里面仅仅输出信息，进程不会被终止

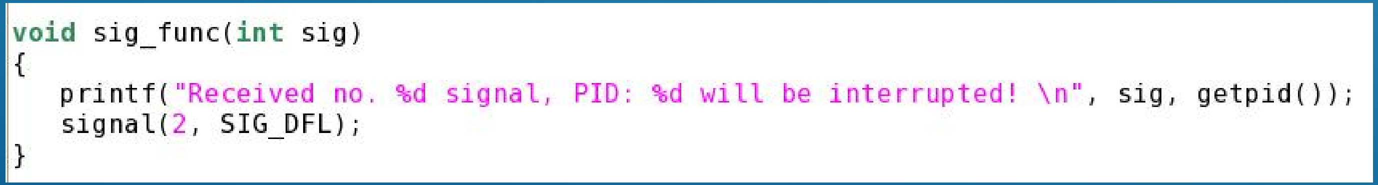


```
void sig_func(int sig)
{
   printf("Received no. %d signal, PID: %d will be interrupted! \n", sig, getpid());
}
```



```
[root@localhost myex]# ./sig
Process is running...
Process is running...
Process is running...
Process is running...
Received no. 2 signal, PID: 7493 will be interrupted!
Process is running...
Process is running...
Received no. 2 signal, PID: 7493 will be interrupted!
Process is running...
Process is running...
Received no. 2 signal, PID: 7493 will be interrupted!
Process is running...
Process is running...
Received no. 2 signal, PID: 7493 will be interrupted!
Process is running...
```

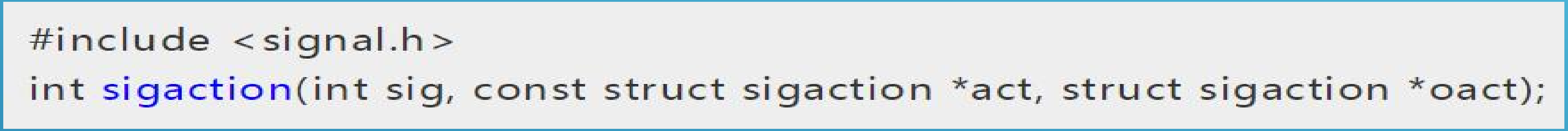
注意2：如果信号处理函数里面，重新恢复缺省处理，观察有什么不一样？



```
void sig_func(int sig)
{
   printf("Received no. %d signal, PID: %d will be interrupted! \n", sig, getpid());
   signal(2, SIG_DFL);
}
```

sigaction信号处理函数

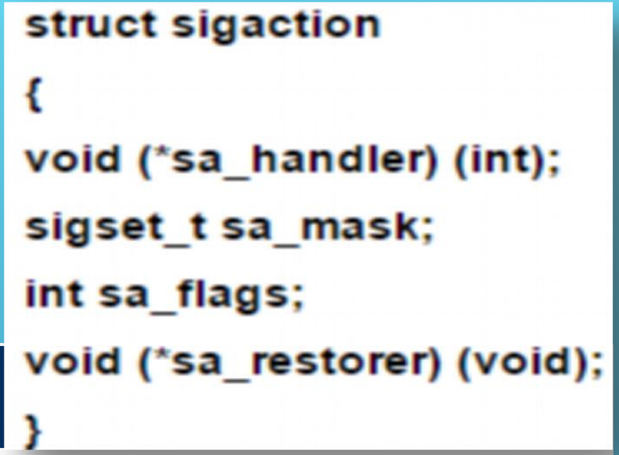
✓ 功能更强的信号接口函数:检查或修改与指定信号相关联的处理动作



```c
#include <signal.h>
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

参数	意义
sig	要改变的信号，SIGKILL,SIGSTOP除外
*act， *oact	如果*act不为空，*oldact不为空，那么oldact将会存储信号以前的行为。 如果*act为空，*oldact不为空，那么oldact将会存储信号现在的行为。

sigaction信号处理函数



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

参数	意义
void (*) (int) sa_handler	SIG_DFL：系统默认处理函数 SIG_IGN：忽略信号 **函数地址**：处理函数地址
sigset_t sa_mask	一个信号集，在调用sa_handler所指向的信号处理函数期间，该信号集中的信号会被阻塞（关）。信号处理函数返回时再将进程的信号屏蔽字复位为原先值（开）。
int sa_flags	改变信号的行为属性： SA_RESETHAND：执行完信号处理函数后重新设为SIG_DFL SA_SIGINFO：需要传递附加信息给信号处理函数，此时使用sa_sigaction成员

信号集（位图）数据类型 sigset_t

- ✓ 一个数组，数组成员从0开始，每个bit对应一个信号；
- ✓ 阻塞信号集：1表示阻塞，0表示不阻塞；
- ✓ 未决信号集：1表示未决信号，0表示无未决信号；

信号集操作函数（修改对应数据的比特位实现信号屏蔽/解除屏蔽）：

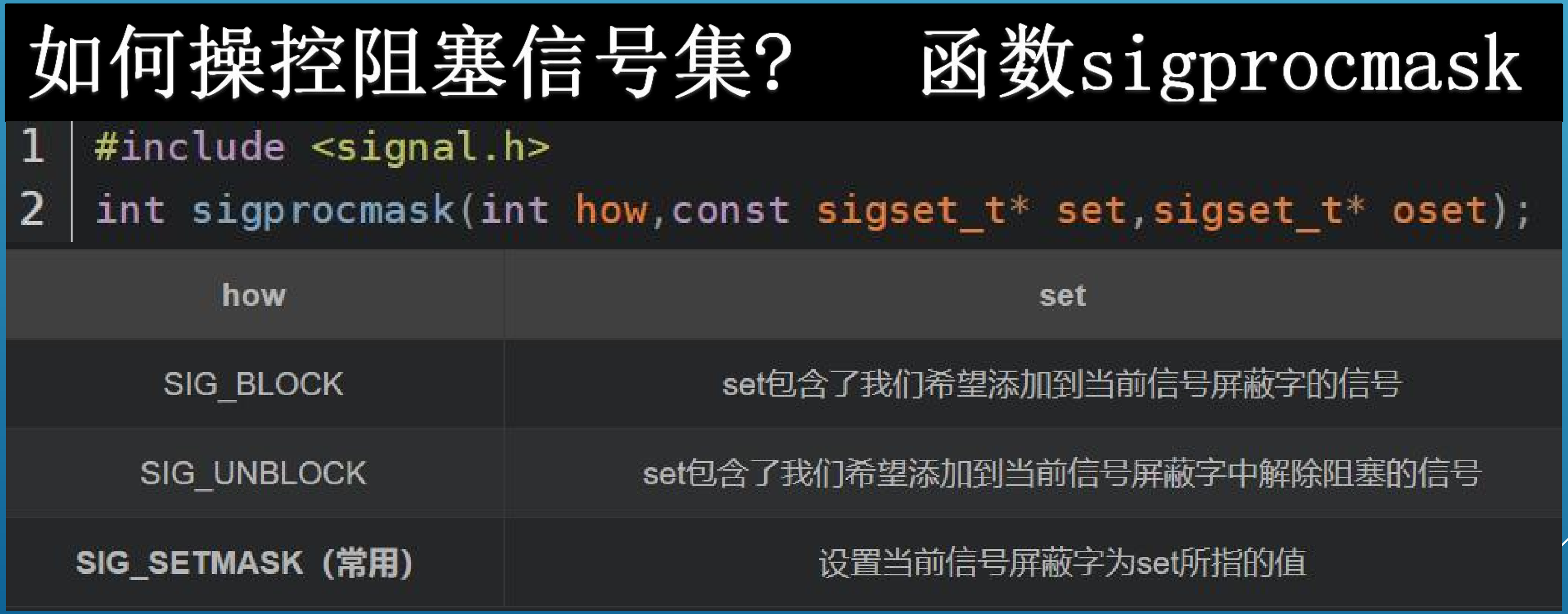
- sigemptyset（sigset_t *set）；
  - 初始化set所指向的信号集，所有比特位清零
- sigfillset（sigset_t *set）
  - 初始化set所指向的信号集，所有比特位置为1
- sigaddset（sigset_t *set， int signo）
  - 对set所指向的信号集添加信号signo，对应位置为1

信号集操作函数（续）：

- sigdelset（sigset_t *set, int signo）
    - 从set所指向的信号集中删除信号signo，对应位置为0
- sigismember（const sigset_t *set, int signo）
    - 用于判断一个信号集的有效信号是否包含signo这个信号
- sigpending(sigset_t *set)
    - 用于读取当前进程的未决信号集

信号屏蔽设置

- ✓ 除了SIGSTOP和SIGKILL外的其他信号都可以暂时屏蔽
- ✓ 即被屏蔽的信号被暂时放置在未决队列中；
- ✓ 对不可靠信号，某种信号不管发送多少，都只有一个未决标志；
- ✓ sigprocmask设置和读取当前进程的屏蔽信号集合

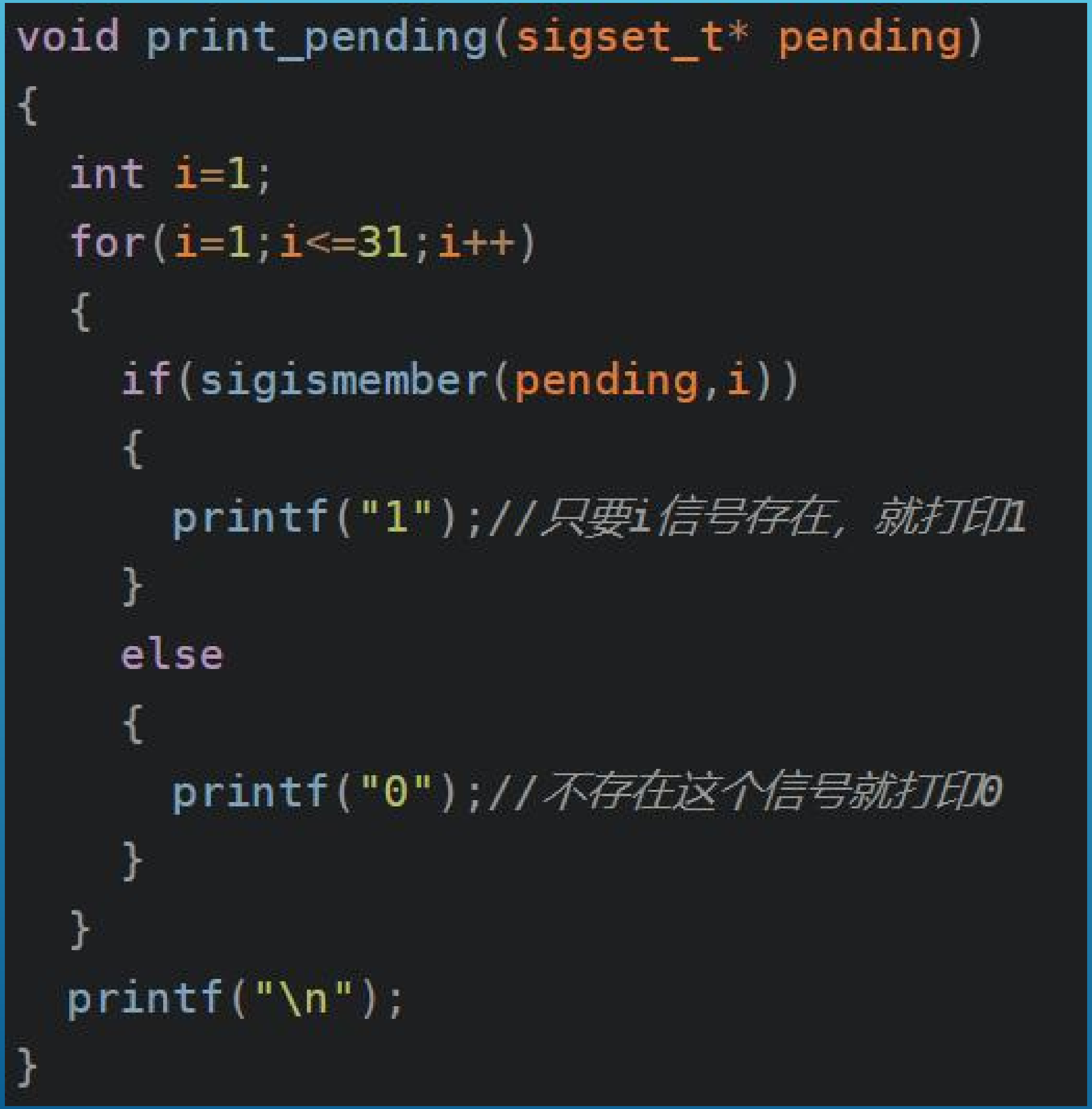


如何操控阻塞信号集?　　函数sigprocmask

```
#include <signal.h>
int sigprocmask(int how,const sigset_t* set,sigset_t* oset);
```

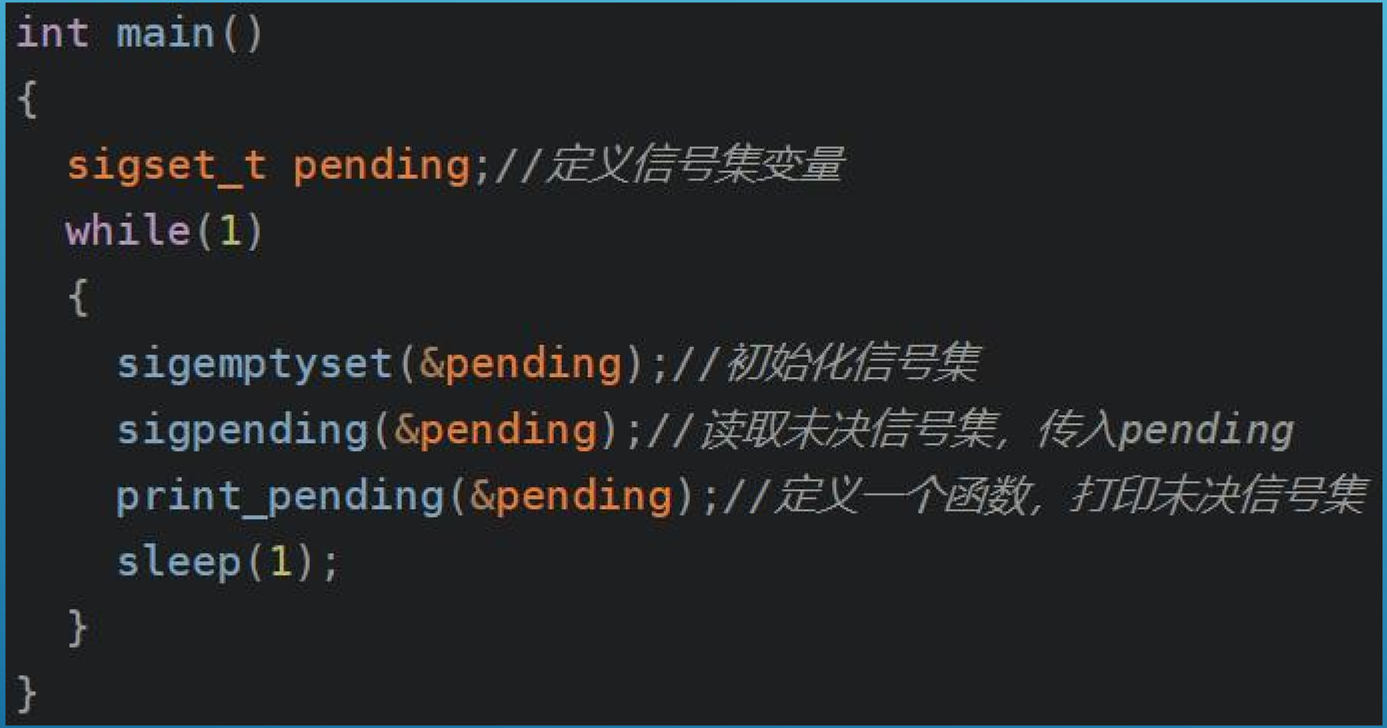
how	set
SIG_BLOCK	set包含了我们希望添加到当前信号屏蔽字的信号
SIG_UNBLOCK	set包含了我们希望添加到当前信号屏蔽字中解除阻塞的信号
SIG_SETMASK（常用）	设置当前信号屏蔽字为set所指的值

打印输出未决信号状态

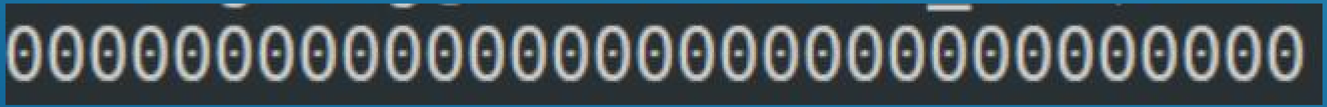


```
void print_pending(sigset_t* pending)
{
  int i=1;
  for(i=1;i<=31;i++)
  {
    if(sigismember(pending,i))
    {
      printf("1");//只要i信号存在，就打印1
    }
    else
    {
      printf("0");//不存在这个信号就打印0
    }
  }
  printf("\n");
}
```

查看未决信号状态

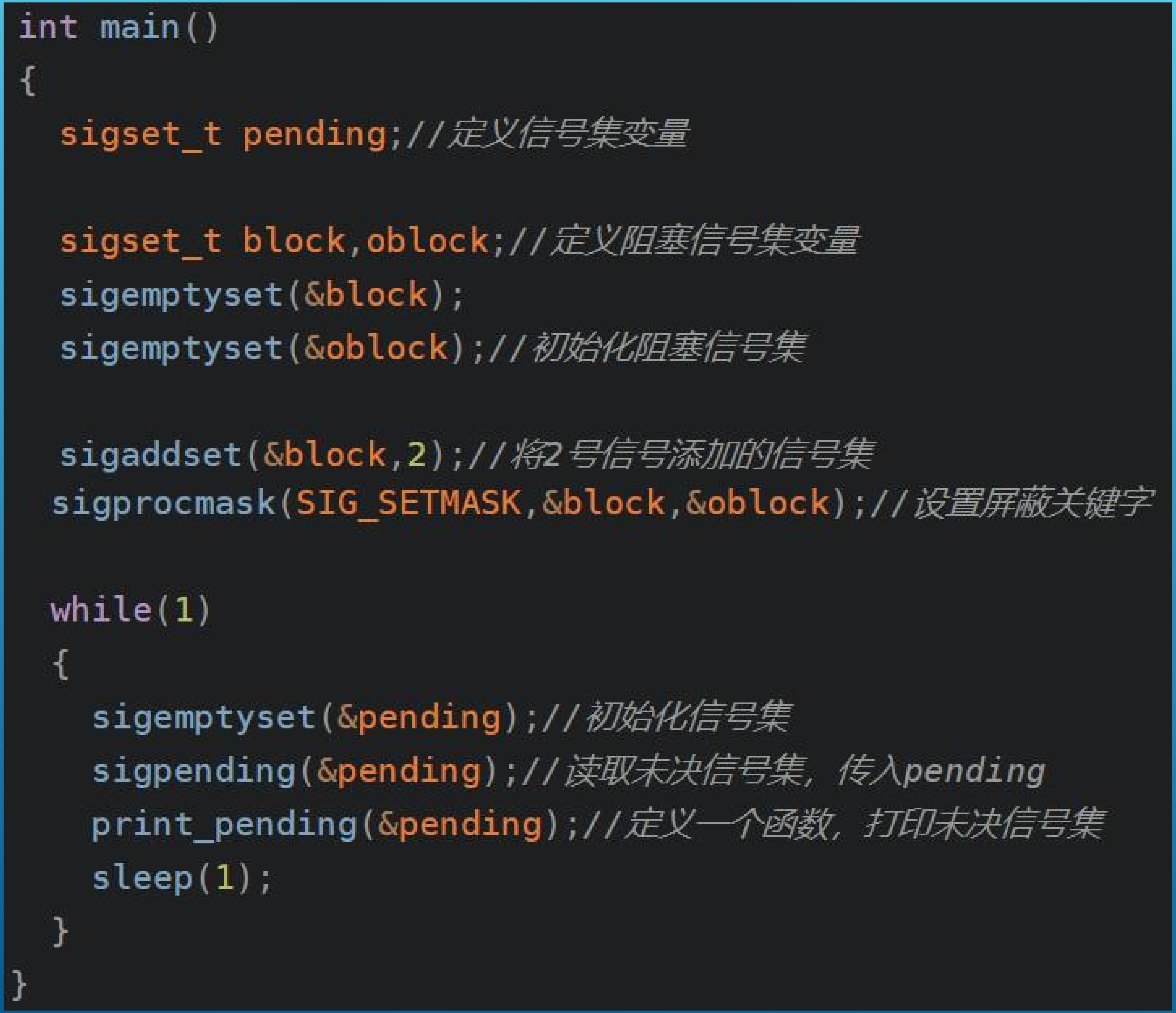


```
int main()
{
  sigset_t pending;//定义信号集变量
  while(1)
  {
    sigemptyset(&pending);//初始化信号集
    sigpending(&pending);//读取未决信号集，传入pending
    print_pending(&pending);//定义一个函数，打印未决信号集
    sleep(1);
  }
}
```



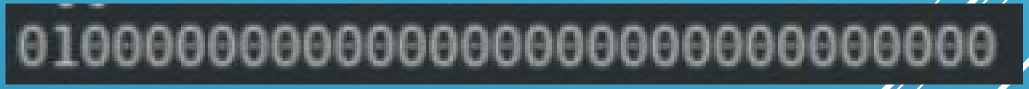
```
0000000000000000000000000000000
```

屏蔽信号2：SIGINT



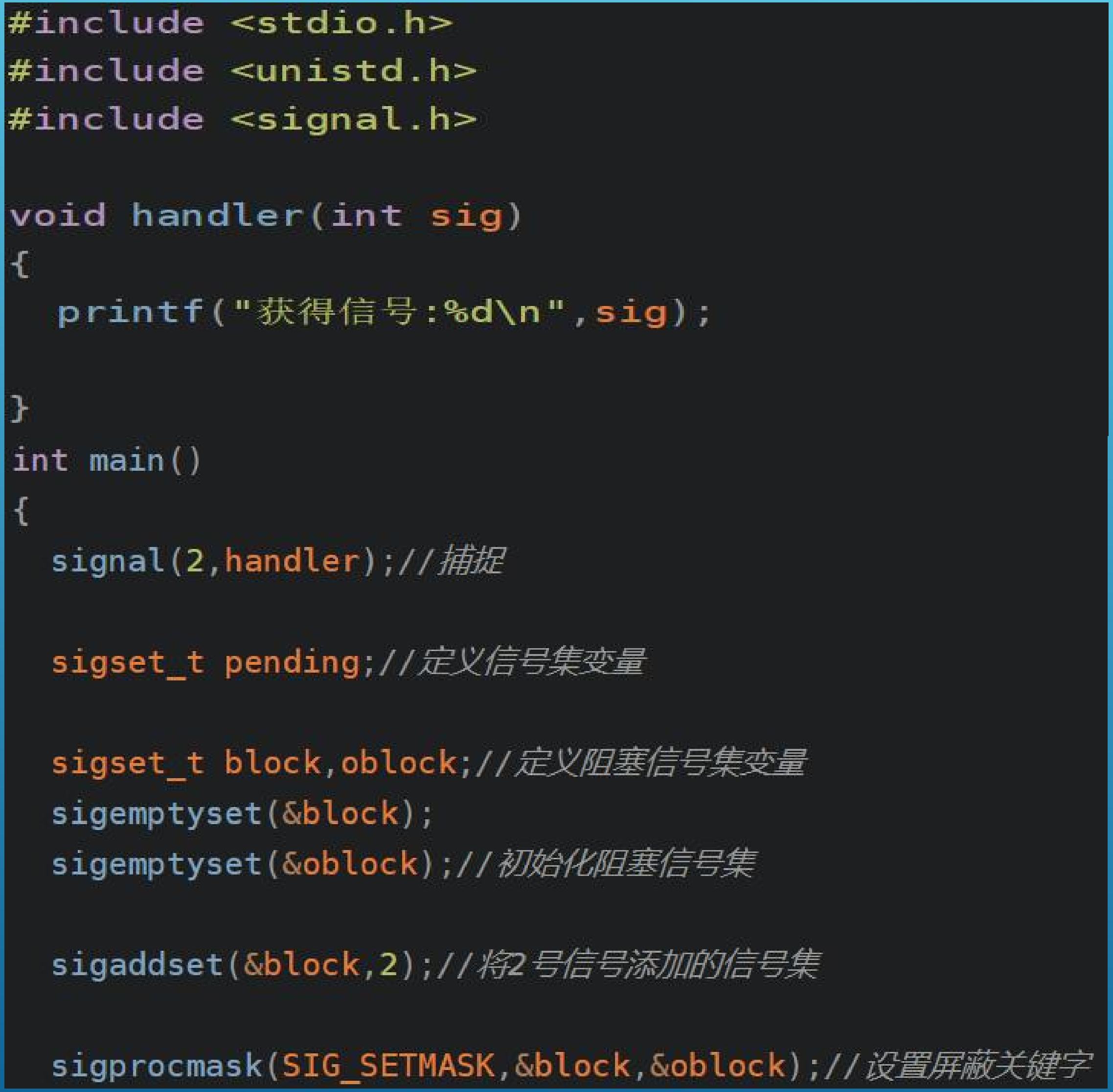
```c
int main()
{
  sigset_t pending;//定义信号集变量

  sigset_t block,oblock;//定义阻塞信号集变量
  sigemptyset(&block);
  sigemptyset(&oblock);//初始化阻塞信号集

  sigaddset(&block,2);//将2号信号添加的信号集
 sigprocmask(SIG_SETMASK,&block,&oblock);//设置屏蔽关键字

 while(1)
 {
   sigemptyset(&pending);//初始化信号集
   sigpending(&pending);//读取未决信号集，传入pending
   print_pending(&pending);//定义一个函数，打印未决信号集
   sleep(1);
 }
}
```

```
$ kill -2  pid
```

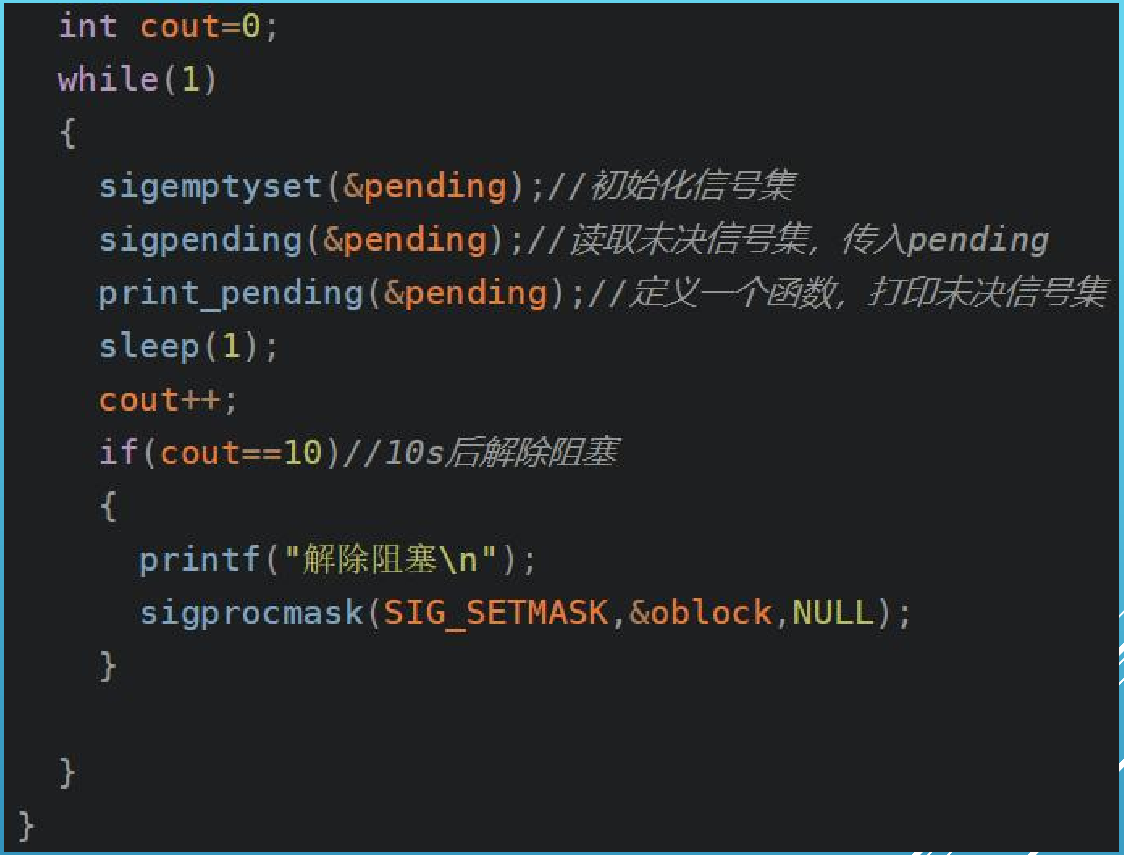


```
0100000000000000000000000000000
```

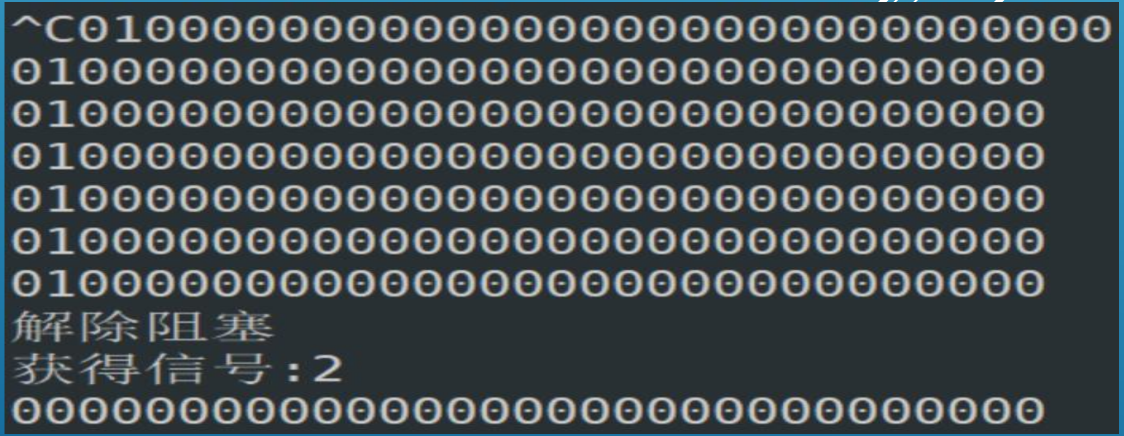
解除屏蔽信号2：SIGINT



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

void handler(int sig)
{
  printf("获得信号:%d\n",sig);

}
int main()
{
  signal(2,handler);//捕捉

  sigset_t pending;//定义信号集变量

  sigset_t block,oblock;//定义阻塞信号集变量
  sigemptyset(&block);
  sigemptyset(&oblock);//初始化阻塞信号集

  sigaddset(&block,2);//将2号信号添加的信号集

  sigprocmask(SIG_SETMASK,&block,&oblock);//设置屏蔽关键字
```



```c
  int cout=0;
  while(1)
  {
    sigemptyset(&pending);//初始化信号集
    sigpending(&pending);//读取未决信号集，传入pending
    print_pending(&pending);//定义一个函数，打印未决信号集
    sleep(1);
    cout++;
    if(cout==10)//10s后解除阻塞
    {
      printf("解除阻塞\n");
      sigprocmask(SIG_SETMASK,&oblock,NULL);
    }

  }
}
```



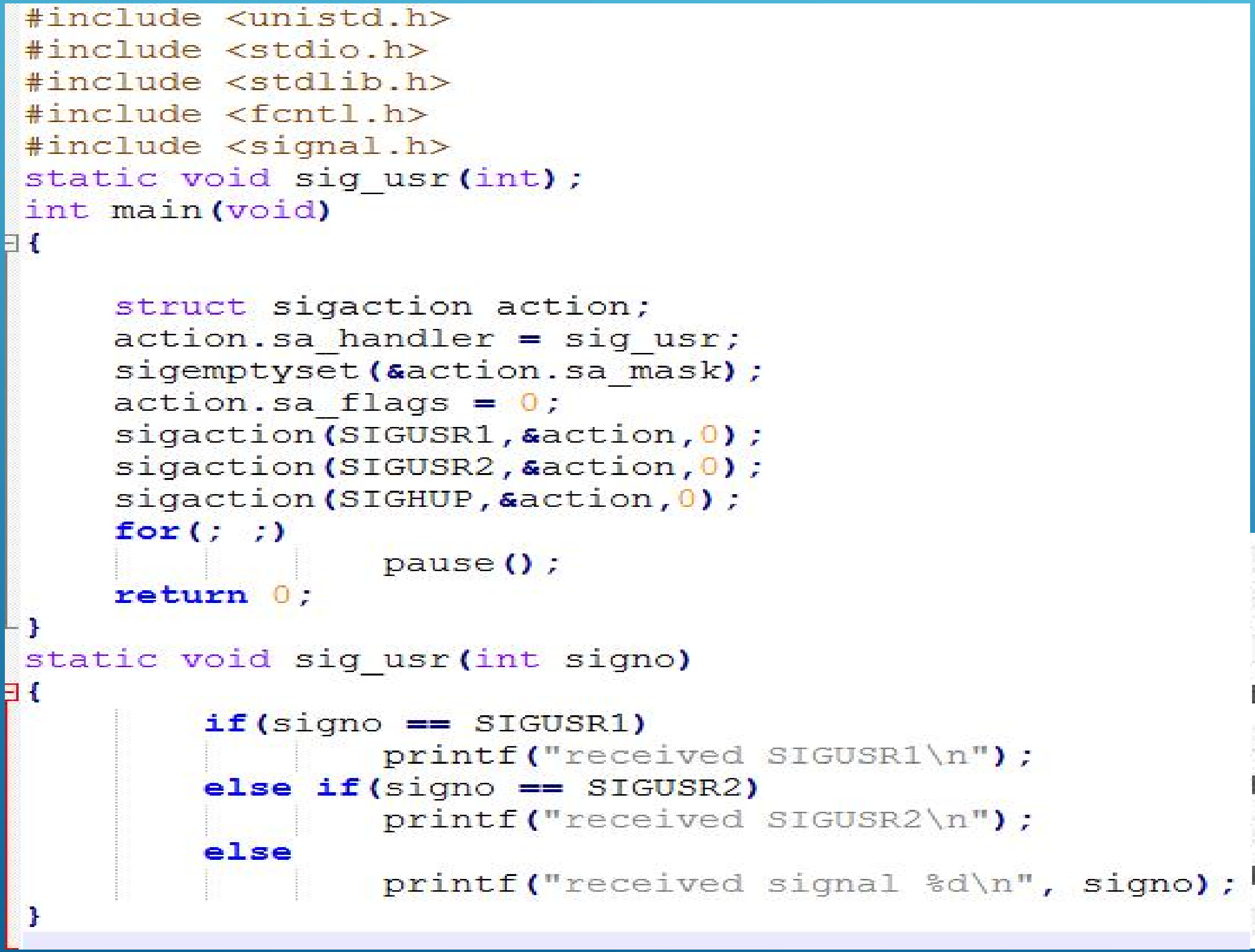
```
^C0100000000000000000000000000000
0100000000000000000000000000000
0100000000000000000000000000000
0100000000000000000000000000000
0100000000000000000000000000000
0100000000000000000000000000000
0100000000000000000000000000000
解除阻塞
获得信号:2
0000000000000000000000000000000
```

信号等待

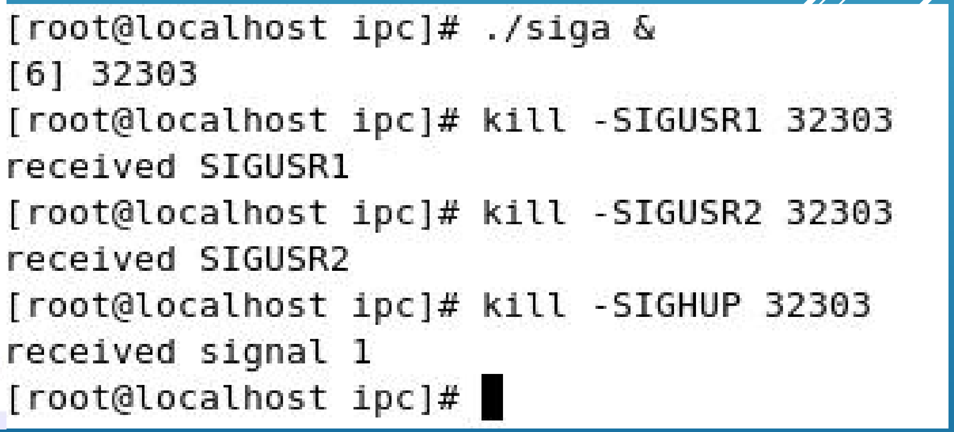
- ✓ 一个进程可以阻塞式等待信号到来
- ✓ Pause阻塞等待任意信号到来
- ✓ Sleep等待一定时间信号到来，但受到其它信号干扰，sleep不再等待

- ✓ `int sigsuspend(const sigset_t *mask);`
- ✓ **阻塞式等待除了mask中屏蔽的信号之外的任意信号；函数返回后，无论之前屏蔽了多少信号，进程屏蔽的信号会切换回去**

SIGACTION安装例1



```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
static void sig_usr(int);
int main(void)
{

    struct sigaction action;
    action.sa_handler = sig_usr;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGUSR1,&action,0);
    sigaction(SIGUSR2,&action,0);
    sigaction(SIGHUP,&action,0);
    for(; ;)
            pause();
    return 0;
}
static void sig_usr(int signo)
{
        if(signo == SIGUSR1)
            printf("received SIGUSR1\n");
        else if(signo == SIGUSR2)
            printf("received SIGUSR2\n");
        else
            printf("received signal %d\n", signo);
}
```



```
[root@localhost ipc]# ./siga &
[6] 32303
[root@localhost ipc]# kill -SIGUSR1 32303
received SIGUSR1
[root@localhost ipc]# kill -SIGUSR2 32303
received SIGUSR2
[root@localhost ipc]# kill -SIGHUP 32303
received signal 1
[root@localhost ipc]# 
```