**一、信号及信号来源**

**信号本质**

信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。

信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。信号机制经过POSIX实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。

**信号来源**

信号事件的发生有两个来源：硬件来源(比如我们按下了键盘或者其它硬件故障)；软件来源，最常用发送信号的系统函数是kill, raise, alarm和setitimer以及sigqueue函数，软件来源还包括一些非法运算等操作。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index1.html#ibm-pcon)

**二、信号的种类**

可以从两个不同的分类角度对信号进行分类：（1）可靠性方面：可靠信号与不可靠信号；（2）与时间的关系上：实时信号与非实时信号。在《Linux环境进程间通信（一）：管道及有名管道》的附1中列出了系统所支持的所有信号。

**1、可靠信号与不可靠信号**

**"不可靠信号"**

Linux信号机制基本上是从Unix系统中继承过来的。早期Unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，因此，把那些建立在早期机制上的信号叫做"不可靠信号"，信号值小于SIGRTMIN(Red hat 7.2中，SIGRTMIN=32，SIGRTMAX=63)的信号都是不可靠信号。这就是"不可靠信号"的来源。它的主要问题是：

* 进程每次处理信号后，就将对信号的响应设置为默认动作。在某些情况下，将导致对信号的错误处理；因此，用户如果不希望这样的操作，那么就要在信号处理函数结尾再一次调用signal()，重新安装该信号。
* 信号可能丢失，后面将对此详细阐述。   
  因此，早期unix下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失。

Linux支持不可靠信号，但是对不可靠信号机制做了改进：在调用完信号处理函数后，不必重新调用该信号的安装函数（信号安装函数是在可靠机制上的实现）。因此，Linux下的不可靠信号问题主要指的是信号可能丢失。

**"可靠信号"**

随着时间的发展，实践证明了有必要对信号的原始机制加以改进和扩充。所以，后来出现的各种Unix版本分别在这方面进行了研究，力图实现"可靠信号"。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失。同时，信号的发送和安装也出现了新版本：信号发送函数sigqueue()及信号安装函数sigaction()。POSIX.4对可靠信号机制做了标准化。但是，POSIX只对可靠信号机制应具有的功能以及信号机制的对外接口做了标准化，对信号机制的实现没有作具体的规定。

信号值位于SIGRTMIN和SIGRTMAX之间的信号都是可靠信号，可靠信号克服了信号可能丢失的问题。Linux在支持新版本的信号安装函数sigation（）以及信号发送函数sigqueue()的同时，仍然支持早期的signal（）信号安装函数，支持信号发送函数kill()。

注：不要有这样的误解：由sigqueue()发送、sigaction安装的信号就是可靠的。事实上，可靠信号是指后来添加的新信号（信号值位于SIGRTMIN及SIGRTMAX之间）；不可靠信号是信号值小于SIGRTMIN的信号。信号的可靠与不可靠只与信号值有关，与信号的发送及安装函数无关。目前linux中的signal()是通过sigation()函数实现的，因此，即使通过signal（）安装的信号，在信号处理函数的结尾也不必再调用一次信号安装函数。同时，由signal()安装的实时信号支持排队，同样不会丢失。

对于目前linux的两个信号安装函数:signal()及sigaction()来说，它们都不能把SIGRTMIN以前的信号变成可靠信号（都不支持排队，仍有可能丢失，仍然是不可靠信号），而且对SIGRTMIN以后的信号都支持排队。这两个函数的最大区别在于，经过sigaction安装的信号都能传递信息给信号处理函数（对所有信号这一点都成立），而经过signal安装的信号却不能向信号处理函数传递信息。对于信号发送函数来说也是一样的。

**2、实时信号与非实时信号**

早期Unix系统只定义了32种信号，Ret hat7.2支持64种信号，编号0-63(SIGRTMIN=31，SIGRTMAX=63)，将来可能进一步增加，这需要得到内核的支持。前32种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的CTRL ^C时，会产生SIGINT信号，对该信号的默认反应就是进程终止。后32个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收。实时信号是POSIX标准的一部分，可用于应用进程。

非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index1.html#ibm-pcon)

**三、进程对信号的响应**

进程可以通过三种方式来响应一个信号：（1）忽略信号，即对信号不做任何处理，其中，有两个信号不能忽略：SIGKILL及SIGSTOP；（2）捕捉信号。定义信号处理函数，当信号发生时，执行相应的处理函数；（3）执行缺省操作，Linux对每种信号都规定了默认操作，详细情况请参考[2]以及其它资料。注意，进程对实时信号的缺省反应是进程终止。

Linux究竟采用上述三种方式的哪一个来响应信号，取决于传递给相应API函数的参数。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index1.html#ibm-pcon)

**四、信号的发送**

发送信号的主要函数有：kill()、raise()、 sigqueue()、alarm()、setitimer()以及abort()。

1、kill()   
#include <sys/types.h>   
#include <signal.h>   
int kill(pid\_t pid,int signo)

|  |  |
| --- | --- |
| 参数pid的值 | 信号的接收进程 |
| pid>0 | 进程ID为pid的进程 |
| pid=0 | 同一个进程组的进程 |
| pid<0 pid!=-1 | 进程组ID为 -pid的所有进程 |
| pid=-1 | 除发送进程自身外，所有进程ID大于1的进程 |

Sinno是信号值，当为0时（即空信号），实际不发送任何信号，但照常进行错误检查，因此，可用于检查目标进程是否存在，以及当前进程是否具有向目标发送信号的权限（root权限的进程可以向任何进程发送信号，非root权限的进程只能向属于同一个session或者同一个用户的进程发送信号）。

Kill()最常用于pid>0时的信号发送，调用成功返回 0； 否则，返回 -1。 注：对于pid<0时的情况，对于哪些进程将接受信号，各种版本说法不一，其实很简单，参阅内核源码kernal/signal.c即可，上表中的规则是参考red hat 7.2。

2、raise（）   
#include <signal.h>   
int raise(int signo)   
向进程本身发送信号，参数为即将发送的信号值。调用成功返回 0；否则，返回 -1。

3、sigqueue（）   
#include <sys/types.h>   
#include <signal.h>   
int sigqueue(pid\_t pid, int sig, const union sigval val)   
调用成功返回 0；否则，返回 -1。

sigqueue()是比较新的发送信号系统调用，主要是针对实时信号提出的（当然也支持前32种），支持信号带有参数，与函数sigaction()配合使用。

sigqueue的第一个参数是指定接收信号的进程ID，第二个参数确定即将发送的信号，第三个参数是一个联合数据结构union sigval，指定了信号传递的参数，即通常所说的4字节值。

|  |
| --- |
| typedef union sigval {  int sival\_int;  void \*sival\_ptr;  }sigval\_t; |

sigqueue()比kill()传递了更多的附加信息，但sigqueue()只能向一个进程发送信号，而不能发送信号给一个进程组。如果signo=0，将会执行错误检查，但实际上不发送任何信号，0值信号可用于检查pid的有效性以及当前进程是否有权限向目标进程发送信号。

在调用sigqueue时，sigval\_t指定的信息会拷贝到3参数信号处理函数（3参数信号处理函数指的是信号处理函数由sigaction安装，并设定了sa\_sigaction指针，稍后将阐述）的siginfo\_t结构中，这样信号处理函数就可以处理这些信息了。由于sigqueue系统调用支持发送带参数信号，所以比kill()系统调用的功能要灵活和强大得多。

注：sigqueue（）发送非实时信号时，第三个参数包含的信息仍然能够传递给信号处理函数； sigqueue（）发送非实时信号时，仍然不支持排队，即在信号处理函数执行过程中到来的所有相同信号，都被合并为一个信号。

4、alarm（）   
#include <unistd.h>   
unsigned int alarm(unsigned int seconds)   
专门为SIGALRM信号而设，在指定的时间seconds秒后，将向进程本身发送SIGALRM信号，又称为闹钟时间。进程调用alarm后，任何以前的alarm()调用都将无效。如果参数seconds为零，那么进程内将不再包含任何闹钟时间。   
返回值，如果调用alarm（）前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回0。

5、setitimer（）   
#include <sys/time.h>   
int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue));   
setitimer()比alarm功能强大，支持3种类型的定时器：

* ITIMER\_REAL： 设定绝对时间；经过指定的时间后，内核将发送SIGALRM信号给本进程；
* ITIMER\_VIRTUAL 设定程序执行时间；经过指定的时间后，内核将发送SIGVTALRM信号给本进程；
* ITIMER\_PROF 设定进程执行以及内核因本进程而消耗的时间和，经过指定的时间后，内核将发送ITIMER\_VIRTUAL信号给本进程；

Setitimer()第一个参数which指定定时器类型（上面三种之一）；第二个参数是结构itimerval的一个实例，结构itimerval形式见附录1。第三个参数可不做处理。

Setitimer()调用成功返回0，否则返回-1。

6、abort()   
#include <stdlib.h>   
void abort(void);

向进程发送SIGABORT信号，默认情况下进程会异常退出，当然可定义自己的信号处理函数。即使SIGABORT被进程设置为阻塞信号，调用abort()后，SIGABORT仍然能被进程接收。该函数无返回值。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index1.html#ibm-pcon)

**五、信号的安装（设置信号关联动作）**

如果进程要处理某一信号，那么就要在进程中安装该信号。安装信号主要用来确定信号值及进程针对该信号值的动作之间的映射关系，即进程将要处理哪个信号；该信号被传递给进程时，将执行何种操作。

linux主要有两个函数实现信号的安装：signal()、sigaction()。其中signal()在可靠信号系统调用的基础上实现, 是库函数。它只有两个参数，不支持信号传递信息，主要是用于前32种非实时信号的安装；而sigaction()是较新的函数（由两个系统调用实现：sys\_signal以及sys\_rt\_sigaction），有三个参数，支持信号传递信息，主要用来与 sigqueue() 系统调用配合使用，当然，sigaction()同样支持非实时信号的安装。sigaction()优于signal()主要体现在支持信号带有参数。

1、signal()   
#include <signal.h>   
void (\*signal(int signum, void (\*handler))(int)))(int);   
如果该函数原型不容易理解的话，可以参考下面的分解方式来理解：   
typedef void (\*sighandler\_t)(int)；   
sighandler\_t signal(int signum, sighandler\_t handler));   
第一个参数指定信号的值，第二个参数指定针对前面信号值的处理，可以忽略该信号（参数设为SIG\_IGN）；可以采用系统默认方式处理信号(参数设为SIG\_DFL)；也可以自己实现处理方式(参数指定一个函数地址)。   
如果signal()调用成功，返回最后一次为安装信号signum而调用signal()时的handler值；失败则返回SIG\_ERR。

2、sigaction()   
#include <signal.h>   
int sigaction(int signum,const struct sigaction \*act,struct sigaction \*oldact));

sigaction函数用于改变进程接收到特定信号后的行为。该函数的第一个参数为信号的值，可以为除SIGKILL及SIGSTOP外的任何一个特定有效的信号（为这两个信号定义自己的处理函数，将导致信号安装错误）。第二个参数是指向结构sigaction的一个实例的指针，在结构sigaction的实例中，指定了对特定信号的处理，可以为空，进程会以缺省方式对信号处理；第三个参数oldact指向的对象用来保存原来对相应信号的处理，可指定oldact为NULL。如果把第二、第三个参数都设为NULL，那么该函数可用于检查信号的有效性。

第二个参数最为重要，其中包含了对指定信号的处理、信号所传递的信息、信号处理函数执行过程中应屏蔽掉哪些函数等等。

sigaction结构定义如下：

|  |
| --- |
| struct sigaction {  union{  \_\_sighandler\_t \_sa\_handler;  void (\*\_sa\_sigaction)(int,struct siginfo \*, void \*)；  }\_u  sigset\_t sa\_mask；  unsigned long sa\_flags；  void (\*sa\_restorer)(void)；  } |

其中，sa\_restorer，已过时，POSIX不支持它，不应再被使用。

1、联合数据结构中的两个元素\_sa\_handler以及\*\_sa\_sigaction指定信号关联函数，即用户指定的信号处理函数。除了可以是用户自定义的处理函数外，还可以为SIG\_DFL(采用缺省的处理方式)，也可以为SIG\_IGN（忽略信号）。

2、由\_sa\_handler指定的处理函数只有一个参数，即信号值，所以信号不能传递除信号值之外的任何信息；由\_sa\_sigaction是指定的信号处理函数带有三个参数，是为实时信号而设的（当然同样支持非实时信号），它指定一个3参数信号处理函数。第一个参数为信号值，第三个参数没有使用（posix没有规范使用该参数的标准），第二个参数是指向siginfo\_t结构的指针，结构中包含信号携带的数据值，参数所指向的结构如下：

|  |
| --- |
| siginfo\_t {  int si\_signo; /\* 信号值，对所有信号有意义\*/  int si\_errno; /\* errno值，对所有信号有意义\*/  int si\_code; /\* 信号产生的原因，对所有信号有意义\*/  union{ /\* 联合数据结构，不同成员适应不同信号 \*/  //确保分配足够大的存储空间  int \_pad[SI\_PAD\_SIZE];  //对SIGKILL有意义的结构  struct{  ...  }...  ... ...  ... ...  //对SIGILL, SIGFPE, SIGSEGV, SIGBUS有意义的结构  struct{  ...  }...  ... ...  }  } |

注：为了更便于阅读，在说明问题时常把该结构表示为附录2所表示的形式。

siginfo\_t结构中的联合数据成员确保该结构适应所有的信号，比如对于实时信号来说，则实际采用下面的结构形式：

|  |
| --- |
| typedef struct {  int si\_signo;  int si\_errno;  int si\_code;  union sigval si\_value;  } siginfo\_t; |

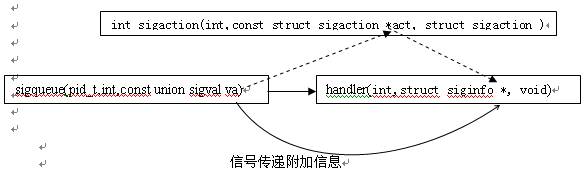
结构的第四个域同样为一个联合数据结构：

|  |
| --- |
| union sigval {  int sival\_int;  void \*sival\_ptr;  } |

采用联合数据结构，说明siginfo\_t结构中的si\_value要么持有一个4字节的整数值，要么持有一个指针，这就构成了与信号相关的数据。在信号的处理函数中，包含这样的信号相关数据指针，但没有规定具体如何对这些数据进行操作，操作方法应该由程序开发人员根据具体任务事先约定。

前面在讨论系统调用sigqueue发送信号时，sigqueue的第三个参数就是sigval联合数据结构，当调用sigqueue时，该数据结构中的数据就将拷贝到信号处理函数的第二个参数中。这样，在发送信号同时，就可以让信号传递一些附加信息。信号可以传递信息对程序开发是非常有意义的。

信号参数的传递过程可图示如下：



3、sa\_mask指定在信号处理程序执行过程中，哪些信号应当被阻塞。缺省情况下当前信号本身被阻塞，防止信号的嵌套发送，除非指定SA\_NODEFER或者SA\_NOMASK标志位。

注：请注意sa\_mask指定的信号阻塞的前提条件，是在由sigaction（）安装信号的处理函数执行过程中由sa\_mask指定的信号才被阻塞。

4、sa\_flags中包含了许多标志位，包括刚刚提到的SA\_NODEFER及SA\_NOMASK标志位。另一个比较重要的标志位是SA\_SIGINFO，当设定了该标志位时，表示信号附带的参数可以被传递到信号处理函数中，因此，应该为sigaction结构中的sa\_sigaction指定处理函数，而不应该为sa\_handler指定信号处理函数，否则，设置该标志变得毫无意义。即使为sa\_sigaction指定了信号处理函数，如果不设置SA\_SIGINFO，信号处理函数同样不能得到信号传递过来的数据，在信号处理函数中对这些信息的访问都将导致段错误（Segmentation fault）。

注：很多文献在阐述该标志位时都认为，如果设置了该标志位，就必须定义三参数信号处理函数。实际不是这样的，验证方法很简单：自己实现一个单一参数信号处理函数，并在程序中设置该标志位，可以察看程序的运行结果。实际上，可以把该标志位看成信号是否传递参数的开关，如果设置该位，则传递参数；否则，不传递参数。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index1.html#ibm-pcon)

**六、信号集及信号集操作函数：**

信号集被定义为一种数据类型：

|  |
| --- |
| typedef struct {  unsigned long sig[\_NSIG\_WORDS]；  } sigset\_t |

信号集用来描述信号的集合，linux所支持的所有信号可以全部或部分的出现在信号集中，主要与信号阻塞相关函数配合使用。下面是为信号集操作定义的相关函数：

|  |
| --- |
| #include <signal.h>  int sigemptyset(sigset\_t \*set)；  int sigfillset(sigset\_t \*set)；  int sigaddset(sigset\_t \*set, int signum)  int sigdelset(sigset\_t \*set, int signum)；  int sigismember(const sigset\_t \*set, int signum)；  sigemptyset(sigset\_t \*set)初始化由set指定的信号集，信号集里面的所有信号被清空；  sigfillset(sigset\_t \*set)调用该函数后，set指向的信号集中将包含linux支持的64种信号；  sigaddset(sigset\_t \*set, int signum)在set指向的信号集中加入signum信号；  sigdelset(sigset\_t \*set, int signum)在set指向的信号集中删除signum信号；  sigismember(const sigset\_t \*set, int signum)判定信号signum是否在set指向的信号集中。 |

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index1.html#ibm-pcon)

**七、信号阻塞与信号未决:**

每个进程都有一个用来描述哪些信号递送到进程时将被阻塞的信号集，该信号集中的所有信号在递送到进程后都将被阻塞。下面是与信号阻塞相关的几个函数：

|  |
| --- |
| #include <signal.h>  int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset))；  int sigpending(sigset\_t \*set));  int sigsuspend(const sigset\_t \*mask))； |

sigprocmask()函数能够根据参数how来实现对信号集的操作，操作主要有三种：

|  |  |
| --- | --- |
| 参数how | 进程当前信号集 |
| SIG\_BLOCK | 在进程当前阻塞信号集中添加set指向信号集中的信号 |
| SIG\_UNBLOCK | 如果进程阻塞信号集中包含set指向信号集中的信号，则解除对该信号的阻塞 |
| SIG\_SETMASK | 更新进程阻塞信号集为set指向的信号集 |

sigpending(sigset\_t \*set))获得当前已递送到进程，却被阻塞的所有信号，在set指向的信号集中返回结果。

sigsuspend(const sigset\_t \*mask))用于在接收到某个信号之前, 临时用mask替换进程的信号掩码, 并暂停进程执行，直到收到信号为止。sigsuspend 返回后将恢复调用之前的信号掩码。信号处理函数完成后，进程将继续执行。该系统调用始终返回-1，并将errno设置为EINTR。

附录1：结构itimerval：

|  |
| --- |
| struct itimerval {  struct timeval it\_interval; /\* next value \*/  struct timeval it\_value; /\* current value \*/  };  struct timeval {  long tv\_sec; /\* seconds \*/  long tv\_usec; /\* microseconds \*/  }; |

附录2：三参数信号处理函数中第二个参数的说明性描述：

|  |
| --- |
| siginfo\_t {  int si\_signo; /\* 信号值，对所有信号有意义\*/  int si\_errno; /\* errno值，对所有信号有意义\*/  int si\_code; /\* 信号产生的原因，对所有信号有意义\*/  pid\_t si\_pid; /\* 发送信号的进程ID,对kill(2),实时信号以及SIGCHLD有意义 \*/  uid\_t si\_uid; /\* 发送信号进程的真实用户ID，对kill(2),实时信号以及SIGCHLD有意义 \*/  int si\_status; /\* 退出状态，对SIGCHLD有意义\*/  clock\_t si\_utime; /\* 用户消耗的时间，对SIGCHLD有意义 \*/  clock\_t si\_stime; /\* 内核消耗的时间，对SIGCHLD有意义 \*/  sigval\_t si\_value; /\* 信号值，对所有实时有意义，是一个联合数据结构，  /\*可以为一个整数（由si\_int标示，也可以为一个指针，由si\_ptr标示）\*/    void \* si\_addr; /\* 触发fault的内存地址，对SIGILL,SIGFPE,SIGSEGV,SIGBUS 信号有意义\*/  int si\_band; /\* 对SIGPOLL信号有意义 \*/  int si\_fd; /\* 对SIGPOLL信号有意义 \*/  } |

实际上，除了前三个元素外，其他元素组织在一个联合结构中，在联合数据结构中，又根据不同的信号组织成不同的结构。注释中提到的对某种信号有意义指的是，在该信号的处理函数中可以访问这些域来获得与信号相关的有意义的信息，只不过特定信号只对特定信息感兴趣而已。

**一、信号生命周期**

从信号发送到信号处理函数的执行完毕

对于一个完整的信号生命周期(从信号发送到相应的处理函数执行完毕)来说，可以分为三个重要的阶段，这三个阶段由四个重要事件来刻画：信号诞生；信号在进程中注册完毕；信号在进程中的注销完毕；信号处理函数执行完毕。相邻两个事件的时间间隔构成信号生命周期的一个阶段。

http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/3.gif

下面阐述四个事件的实际意义：

1. 信号"诞生"。信号的诞生指的是触发信号的事件发生（如检测到硬件异常、定时器超时以及调用信号发送函数kill()或sigqueue()等）。
2. 信号在目标进程中"注册"；进程的task\_struct结构中有关于本进程中未决信号的数据成员：

|  |
| --- |
| struct sigpending pending：  struct sigpending{  struct sigqueue \*head, \*\*tail;  sigset\_t signal;  }; |

第三个成员是进程中所有未决信号集，第一、第二个成员分别指向一个sigqueue类型的结构链（称之为"未决信号信息链"）的首尾，信息链中的每个sigqueue结构刻画一个特定信号所携带的信息，并指向下一个sigqueue结构:

|  |
| --- |
| struct sigqueue{  struct sigqueue \*next;  siginfo\_t info;  } |

信号在进程中注册指的就是信号值加入到进程的未决信号集中（sigpending结构的第二个成员sigset\_t signal），并且信号所携带的信息被保留到未决信号信息链的某个sigqueue结构中。 只要信号在进程的未决信号集中，表明进程已经知道这些信号的存在，但还没来得及处理，或者该信号被进程阻塞。

**注：**   
当一个实时信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次，因此，信号不会丢失，因此，实时信号又叫做"可靠信号"。这意味着同一个实时信号可以在同一个进程的未决信号信息链中占有多个sigqueue结构（进程每收到一个实时信号，都会为它分配一个结构来登记该信号信息，并把该结构添加在未决信号链尾，即所有诞生的实时信号都会在目标进程中注册）；   
当一个非实时信号发送给一个进程时，如果该信号已经在进程中注册，则该信号将被丢弃，造成信号丢失。因此，非实时信号又叫做"不可靠信号"。这意味着同一个非实时信号在进程的未决信号信息链中，至多占有一个sigqueue结构（一个非实时信号诞生后，（1）、如果发现相同的信号已经在目标结构中注册，则不再注册，对于进程来说，相当于不知道本次信号发生，信号丢失；（2）、如果进程的未决信号中没有相同信号，则在进程中注册自己）。

1. 信号在进程中的注销。在目标进程执行过程中，会检测是否有信号等待处理（每次从系统空间返回到用户空间时都做这样的检查）。如果存在未决信号等待处理且该信号没有被进程阻塞，则在运行相应的信号处理函数前，进程会把信号在未决信号链中占有的结构卸掉。是否将信号从进程未决信号集中删除对于实时与非实时信号是不同的。对于非实时信号来说，由于在未决信号信息链中最多只占用一个sigqueue结构，因此该结构被释放后，应该把信号在进程未决信号集中删除（信号注销完毕）；而对于实时信号来说，可能在未决信号信息链中占用多个sigqueue结构，因此应该针对占用sigqueue结构的数目区别对待：如果只占用一个sigqueue结构（进程只收到该信号一次），则应该把信号在进程的未决信号集中删除（信号注销完毕）。否则，不应该在进程的未决信号集中删除该信号（信号注销完毕）。   
   进程在执行信号相应处理函数之前，首先要把信号在进程中注销。
2. 信号生命终止。进程注销信号后，立即执行相应的信号处理函数，执行完毕后，信号的本次发送对进程的影响彻底结束。

**注：**   
1）信号注册与否，与发送信号的函数（如kill()或sigqueue()等）以及信号安装函数（signal()及sigaction()）无关，只与信号值有关（信号值小于SIGRTMIN的信号最多只注册一次，信号值在SIGRTMIN及SIGRTMAX之间的信号，只要被进程接收到就被注册）。   
2）在信号被注销到相应的信号处理函数执行完毕这段时间内，如果进程又收到同一信号多次，则对实时信号来说，每一次都会在进程中注册；而对于非实时信号来说，无论收到多少次信号，都会视为只收到一个信号，只在进程中注册一次。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index2.html#ibm-pcon)

**二、信号编程注意事项**

1. 防止不该丢失的信号丢失。如果对八中所提到的信号生命周期理解深刻的话，很容易知道信号会不会丢失，以及在哪里丢失。
2. **程序的可移植性**   
   考虑到程序的可移植性，应该尽量采用POSIX信号函数，POSIX信号函数主要分为两类：
   * POSIX 1003.1信号函数： Kill()、sigaction()、sigaddset()、sigdelset()、sigemptyset()、sigfillset()、sigismember()、sigpending()、sigprocmask()、sigsuspend()。
   * POSIX 1003.1b信号函数。POSIX 1003.1b在信号的实时性方面对POSIX 1003.1做了扩展，包括以下三个函数： sigqueue()、sigtimedwait()、sigwaitinfo()。 其中，sigqueue主要针对信号发送，而sigtimedwait及sigwaitinfo()主要用于取代sigsuspend()函数，后面有相应实例。

|  |
| --- |
| #include <signal.h>  int sigwaitinfo(sigset\_t \*set, siginfo\_t \*info). |

该函数与sigsuspend()类似，阻塞一个进程直到特定信号发生，但信号到来时不执行信号处理函数，而是返回信号值。因此为了避免执行相应的信号处理函数，必须在调用该函数前，使进程屏蔽掉set指向的信号，因此调用该函数的典型代码是：

|  |
| --- |
| sigset\_t newmask;  int rcvd\_sig;  siginfo\_t info;  sigemptyset(&newmask);  sigaddset(&newmask, SIGRTMIN);  sigprocmask(SIG\_BLOCK, &newmask, NULL);  rcvd\_sig = sigwaitinfo(&newmask, &info)  if (rcvd\_sig == -1) {  ..  } |

调用成功返回信号值，否则返回-1。sigtimedwait()功能相似，只不过增加了一个进程等待的时间。

1. **程序的稳定性。**   
   为了增强程序的稳定性，在信号处理函数中应使用可重入函数。

信号处理程序中应当使用可再入（可重入）函数（注：所谓可重入函数是指一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错）。因为进程在收到信号后，就将跳转到信号处理函数去接着执行。如果信号处理函数中使用了不可重入函数，那么信号处理函数可能会修改原来进程中不应该被修改的数据，这样进程从信号处理函数中返回接着执行时，可能会出现不可预料的后果。不可再入函数在信号处理函数中被视为不安全函数。

满足下列条件的函数多数是不可再入的：（1）使用静态的数据结构，如getlogin()，gmtime()，getgrgid()，getgrnam()，getpwuid()以及getpwnam()等等；（2）函数实现时，调用了malloc（）或者free()函数；（3）实现时使用了标准I/O函数的。The Open Group视下列函数为可再入的：

\_exit（）、access（）、alarm（）、cfgetispeed（）、cfgetospeed（）、cfsetispeed（）、cfsetospeed（）、chdir（）、chmod（）、chown（） 、close（）、creat（）、dup（）、dup2（）、execle（）、execve（）、fcntl（）、fork（）、fpathconf（）、fstat（）、fsync（）、getegid（）、 geteuid（）、getgid（）、getgroups（）、getpgrp（）、getpid（）、getppid（）、getuid（）、kill（）、link（）、lseek（）、mkdir（）、mkfifo（）、 open（）、pathconf（）、pause（）、pipe（）、raise（）、read（）、rename（）、rmdir（）、setgid（）、setpgid（）、setsid（）、setuid（）、 sigaction（）、sigaddset（）、sigdelset（）、sigemptyset（）、sigfillset（）、sigismember（）、signal（）、sigpending（）、sigprocmask（）、sigsuspend（）、sleep（）、stat（）、sysconf（）、tcdrain（）、tcflow（）、tcflush（）、tcgetattr（）、tcgetpgrp（）、tcsendbreak（）、tcsetattr（）、tcsetpgrp（）、time（）、times（）、 umask（）、uname（）、unlink（）、utime（）、wait（）、waitpid（）、write（）。

即使信号处理函数使用的都是"安全函数"，同样要注意进入处理函数时，首先要保存errno的值，结束时，再恢复原值。因为，信号处理过程中，errno值随时可能被改变。另外，longjmp()以及siglongjmp()没有被列为可再入函数，因为不能保证紧接着两个函数的其它调用是安全的。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index2.html#ibm-pcon)

**三、深入浅出：信号应用实例**

linux下的信号应用并没有想象的那么恐怖，程序员所要做的最多只有三件事情：

1. 安装信号（推荐使用sigaction()）；
2. 实现三参数信号处理函数，handler(int signal,struct siginfo \*info, void \*)；
3. 发送信号，推荐使用sigqueue()。

实际上，对有些信号来说，只要安装信号就足够了（信号处理方式采用缺省或忽略）。其他可能要做的无非是与信号集相关的几种操作。

**实例一：信号发送及处理**   
实现一个信号接收程序sigreceive（其中信号安装由sigaction（））。

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  void new\_op(int,siginfo\_t\*,void\*);  int main(int argc,char\*\*argv)  {  struct sigaction act;  int sig;  sig=atoi(argv[1]);    sigemptyset(&act.sa\_mask);  act.sa\_flags=SA\_SIGINFO;  act.sa\_sigaction=new\_op;    if(sigaction(sig,&act,NULL) < 0)  {  printf("install sigal error\n");  }    while(1)  {  sleep(2);  printf("wait for the signal\n");  }  }  void new\_op(int signum,siginfo\_t \*info,void \*myact)  {  printf("receive signal %d", signum);  sleep(5);  } |

说明，命令行参数为信号值，后台运行sigreceive signo &，可获得该进程的ID，假设为pid，然后再另一终端上运行kill -s signo pid验证信号的发送接收及处理。同时，可验证信号的排队问题。   
**注：**可以用sigqueue实现一个命令行信号发送程序sigqueuesend，见 [附录1](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index2.html#5)。

**实例二：信号传递附加信息**   
主要包括两个实例：

1. 向进程本身发送信号，并传递指针参数；

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  void new\_op(int,siginfo\_t\*,void\*);  int main(int argc,char\*\*argv)  {  struct sigaction act;  union sigval mysigval;  int i;  int sig;  pid\_t pid;  char data[10];  memset(data,0,sizeof(data));  for(i=0;i < 5;i++)  data[i]='2';  mysigval.sival\_ptr=data;    sig=atoi(argv[1]);  pid=getpid();    sigemptyset(&act.sa\_mask);  act.sa\_sigaction=new\_op;//三参数信号处理函数  act.sa\_flags=SA\_SIGINFO;//信息传递开关  if(sigaction(sig,&act,NULL) < 0)  {  printf("install sigal error\n");  }  while(1)  {  sleep(2);  printf("wait for the signal\n");  sigqueue(pid,sig,mysigval);//向本进程发送信号，并传递附加信息  }  }  void new\_op(int signum,siginfo\_t \*info,void \*myact)//三参数信号处理函数的实现  {  int i;  for(i=0;i<10;i++)  {  printf("%c\n ",(\*( (char\*)((\*info).si\_ptr)+i)));  }  printf("handle signal %d over;",signum);  } |

这个例子中，信号实现了附加信息的传递，信号究竟如何对这些信息进行处理则取决于具体的应用。

1. 2、 不同进程间传递整型参数：把1中的信号发送和接收放在两个程序中，并且在发送过程中传递整型参数。   
   信号接收程序：

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  void new\_op(int,siginfo\_t\*,void\*);  int main(int argc,char\*\*argv)  {  struct sigaction act;  int sig;  pid\_t pid;    pid=getpid();  sig=atoi(argv[1]);    sigemptyset(&act.sa\_mask);  act.sa\_sigaction=new\_op;  act.sa\_flags=SA\_SIGINFO;  if(sigaction(sig,&act,NULL)<0)  {  printf("install sigal error\n");  }  while(1)  {  sleep(2);  printf("wait for the signal\n");  }  }  void new\_op(int signum,siginfo\_t \*info,void \*myact)  {  printf("the int value is %d \n",info->si\_int);  } |

信号发送程序：命令行第二个参数为信号值，第三个参数为接收进程ID。

|  |
| --- |
| #include <signal.h>  #include <sys/time.h>  #include <unistd.h>  #include <sys/types.h>  main(int argc,char\*\*argv)  {  pid\_t pid;  int signum;  union sigval mysigval;  signum=atoi(argv[1]);  pid=(pid\_t)atoi(argv[2]);  mysigval.sival\_int=8;//不代表具体含义，只用于说明问题  if(sigqueue(pid,signum,mysigval)==-1)  printf("send error\n");  sleep(2);  } |

**注：**实例2的两个例子侧重点在于用信号来传递信息，目前关于在linux下通过信号传递信息的实例非常少，倒是Unix下有一些，但传递的基本上都是关于传递一个整数，传递指针的我还没看到。我一直没有实现不同进程间的指针传递（实际上更有意义），也许在实现方法上存在问题吧，请实现者email我。

**实例三：信号阻塞及信号集操作**

|  |
| --- |
| #include "signal.h"  #include "unistd.h"  static void my\_op(int);  main()  {  sigset\_t new\_mask,old\_mask,pending\_mask;  struct sigaction act;  sigemptyset(&act.sa\_mask);  act.sa\_flags=SA\_SIGINFO;  act.sa\_sigaction=(void\*)my\_op;  if(sigaction(SIGRTMIN+10,&act,NULL))  printf("install signal SIGRTMIN+10 error\n");  sigemptyset(&new\_mask);  sigaddset(&new\_mask,SIGRTMIN+10);  if(sigprocmask(SIG\_BLOCK, &new\_mask,&old\_mask))  printf("block signal SIGRTMIN+10 error\n");  sleep(10);  printf("now begin to get pending mask and unblock SIGRTMIN+10\n");  if(sigpending(&pending\_mask)<0)  printf("get pending mask error\n");  if(sigismember(&pending\_mask,SIGRTMIN+10))  printf("signal SIGRTMIN+10 is pending\n");  if(sigprocmask(SIG\_SETMASK,&old\_mask,NULL)<0)  printf("unblock signal error\n");  printf("signal unblocked\n");  sleep(10);  }  static void my\_op(int signum)  {  printf("receive signal %d \n",signum);  } |

编译该程序，并以后台方式运行。在另一终端向该进程发送信号(运行kill -s 42 pid，SIGRTMIN+10为42)，查看结果可以看出几个关键函数的运行机制，信号集相关操作比较简单。

**注：**在上面几个实例中，使用了printf()函数，只是作为诊断工具，pringf()函数是不可重入的，不应在信号处理函数中使用。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index2.html#ibm-pcon)

**结束语：**

系统地对linux信号机制进行分析、总结使我受益匪浅！感谢王小乐等网友的支持！   
Comments and suggestions are greatly welcome!

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index2.html#ibm-pcon)

**附录1：**

用sigqueue实现的命令行信号发送程序sigqueuesend，命令行第二个参数是发送的信号值，第三个参数是接收该信号的进程ID，可以配合实例一使用：

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  int main(int argc,char\*\*argv)  {  pid\_t pid;  int sig;  sig=atoi(argv[1]);  pid=atoi(argv[2]);  sigqueue(pid,sig,NULL);  sleep(2);  } |

**参考资料**

* linux内核源代码情景分析（上），毛德操、胡希明著，浙江大学出版社，当要验证某个结论、想法时，最好的参考资料；
* UNIX环境高级编程，作者：W.Richard Stevens，译者：尤晋元等，机械工业出版社。对信号机制的发展过程阐述的比较详细。
* signal、sigaction、kill等手册，最直接而可靠的参考资料。
* <http://www.linuxjournal.com/modules.php?op=modload&name=NS-help&file=man>提供了许多系统调用、库函数等的在线指南。
* <http://www.opengroup.org/onlinepubs/007904975/>可以在这里对许多关键函数（包括系统调用）进行查询，非常好的一个网址。
* <http://unix.org/whitepapers/reentrant.html>对函数可重入进行了阐述。
* <http://www.uccs.edu/~compsvcs/doc-cdrom/DOCS/HTML/APS33DTE/DOCU_006.HTM>对实时信号给出了相当好的描述。