Linux信号（signal) 机制分析

【摘要】本文分析了Linux内核对于信号的实现机制和应用层的相关处理。首先介绍了软中断信号的本质及信号的两种不同分类方法尤其是不可靠信号的原理。接着分析了内核对于信号的处理流程包括信号的触发/注册/执行及注销等。最后介绍了应用层的相关处理，主要包括信号处理函数的安装、信号的发送、屏蔽阻塞等，最后给了几个简单的应用实例。

【关键字】软中断信号，`signal`，`sigaction`，`kill`，`sigqueue`，`settimer`，`sigmask`，`sigprocmask`，`sigset_t`

1 信号本质

软中断信号（signal，又简称为信号）用来通知进程发生了异步事件。在软件层次上是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。信号是进程间通信机制中唯一的异步通信机制，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。进程之间可以互相通过系统调用kill发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。信号机制除了基本通知功能外，还可以传递附加信息。

收到信号的进程对各种信号有不同的处理方法。处理方法可以分为三类：

第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。

第二种方法是，忽略某个信号，对该信号不做任何处理，就象未发生过一样。

第三种方法是，对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信号的缺省操作是使得进程终止。进程通过系统调用signal来指定进程对某个信号的处理行为。

2 信号的种类

可以从两个不同的分类角度对信号进行分类：

可靠性方面：可靠信号与不可靠信号；

与时间的关系上：实时信号与非实时信号。

2.1 可靠信号与不可靠信号

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，信号值小于SIGRTMIN（34）的信号都是不可靠信号。这就是"不可靠信号"的来源。它的主要问题是信号可能丢失。

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。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。

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()`。

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，而经过`signal`安装的信号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。

2.2 实时信号与非实时信号

早期Unix系统只定义了32种信号，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。

（找不到32号和33号信号）

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。

3 信号处理流程

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，可以分为三个阶段：

1. 信号诞生
2. 信号在进程中注册
3. 信号的执行和注销

3.1 信号诞生

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。

这里按发出信号的原因简单分类，以了解各种信号：

1. 与进程终止相关的信号。当进程退出，或者子进程终止时，发出这类信号。
2. 与进程例外事件相关的信号。如进程越界，或企图写一个只读的内存区域（如程序正文区），或执行一个特权指令及其他各种硬件错误。
3. 与在系统调用期间遇到不可恢复条件相关的信号。如执行系统调用exec时，原有资源已经释放，而目前系统资源又已经耗尽。
4. 与执行系统调用时遇到非预测错误条件相关的信号。如执行一个并不存在的系统调用。
5. 在用户态下的进程发出的信号。如进程调用系统调用kill向其他进程发送信号。
6. 与终端交互相关的信号。如用户关闭一个终端，或按下break键等情况。
7. 跟踪进程执行的信号。

Linux支持的信号列表如下。很多信号是与机器的体系结构相关的

信号值	默认处理动作	发出信号的原因
SIGHUP 1	A	终端挂起或者控制进程终止
SIGINT 2	A	键盘中断（如break键被按下）
SIGQUIT 3	C	键盘的退出键被按下
SIGILL 4	C	非法指令
SIGABRT 6	C	由abort(3)发出的退出指令
SIGFPE 8	C	浮点异常
SIGKILL 9	AEF	Kill信号
SIGSEGV 11	C	无效的内存引用
SIGPIPE 13	A	管道破裂: 写一个没有读端口的管道
SIGALRM 14	A	由alarm(2)发出的信号
SIGTERM 15	A	终止信号
SIGUSR1 30,10,16	A	用户自定义信号1
SIGUSR2 31,12,17	A	用户自定义信号2
SIGCHLD 20,17,18	B	子进程结束信号
SIGCONT 19,18,25		进程继续（曾被停止的进程）
SIGSTOP 17,19,23	DEF	终止进程
SIGTSTP 18,20,24	D	控制终端（tty）上按下停止键
SIGTTIN 21,21,26	D	后台进程企图从控制终端读
SIGTTOU 22,22,27	D	后台进程企图从控制终端写

处理动作一项中的字母含义如下

码	处理动作
A	缺省的动作是终止进程

码	处理动作
B	缺省的动作是忽略此信号，将该信号丢弃，不做处理
C	缺省的动作是终止进程并进行内核映像转储（dump core），内核映像转储是指将进程数据在内存的映像和进程在内核结构中的部分内容以一定格式转储到文件系统，并且进程退出执行，这样做的好处是为程序员提供了方便，使得他们可以得到进程当时执行时的数据值，允许他们确定转储的原因，并且可以调试他们的程序。
D	缺省的动作是停止进程，进入停止状况以后还能重新进行下去，一般是在调试的过程中（例如ptrace系统调用）
E	信号不能被捕获
F	信号不能被忽略

3.2 信号在目标进程中注册

在进程表[1]的表项中有一个软中断信号域，该域中每一位对应一个信号。内核给一个进程发送软中断信号的方法，是在进程所在的进程表项的信号域设置对应于该信号的位(bit)。如果信号发送给一个正在睡眠的进程，如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位(bit)，而不唤醒进程。如果发送给一个处于可运行状态的进程，则只置相应的域即可。

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`sigset_t signal`（每个信号占用一位）中，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。

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（通过`sigset_t signal`指示），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。

总之信号注册与否，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）

3.3 信号的执行和注销

内核处理一个进程收到的软中断信号是在该进程的上下文中，因此，进程必须处于运行状态。当其由于被信号唤醒或者正常调度重新获得CPU时，在其从内核空间返回到用户空间时会检测是否有信号等待处理。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。

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），则执行完相应的处理函数后应该把信号在进程的未决信号集中删除（信号注销完毕）。否则待该信号的所有`sigqueue`处理完毕后再在进程的未决信号集中删除该信号。

当所有未被屏蔽的信号都处理完毕后，即可返回用户空间。对于被屏蔽的信号，当取消屏蔽后，在返回到用户空间时会再次执行上述检查处理的一套流程。

内核处理一个进程收到的信号的时机是在一个进程从内核态返回用户态时。所以，当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。

处理信号有三种类型：进程接收到信号后退出；进程忽略该信号；进程收到信号后执行用户设定用系统调用`signal`的函数。当进程接收到一个它忽略的信号时，进程丢弃该信号，就象没有收到该信号似的继续运行。如果进程收到一个要捕捉的信号，那么进程从内核态返回用户态时执行用户定义的函数。而且执行用户定义的函数的方法很巧妙，内核是在用户栈上创建一个新的层，该层中将返回地址的值设置成用户定义的处理函数的地址，这样进程从内核返回弹出栈顶时就返回到用户定义的函数处，从函数返回再弹出栈顶时，才返回原先进入内核的地方。这样做的原因是用户定义的处理函数不能且不允许在内核态下执行（如果用户定义的函数在内核态下运行的话，用户就可以获得任何权限）。

4 信号的安装

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()`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。

4.1 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。

传递给信号处理例程的整数参数是信号值，这样可以使得一个信号处理例程处理多个信号。

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

void sigroutine(int dunno)
{ /* 信号处理例程，其中dunno将会得到信号的值 */
    switch (dunno) {
    case 1:
        printf("Get a signal -- SIGHUP ");
        break;
    case 2:
        printf("Get a signal -- SIGINT ");
        break;
    case 3:
        printf("Get a signal -- SIGQUIT ");
        break;
    }
    return;
}

int main() {
    printf("process id is %d ",getpid());
```

```
    signal(SIGHUP, sigroutine); //* 下面设置三个信号的处理方法
    signal(SIGINT, sigroutine);
    signal(SIGQUIT, sigroutine);
    for (;;) ;
}
```

其中信号SIGINT由按下Ctrl-C发出，信号SIGQUIT由按下Ctrl-发出。该程序执行的结果如下：

```
localhost:~$ ./sig_test

process id is 463

Get a signal -SIGINT //按下Ctrl-C得到的结果

Get a signal -SIGQUIT //按下Ctrl-\得到的结果

//按下Ctrl-z将进程置于后台

 [1]+ Stopped ./sig_test

localhost:~$ bg //

 [1]+ ./sig_test &

localhost:~$ kill -HUP 463 //向进程发送SIGHUP信号

localhost:~$ Get a signal - SIGHUP

kill -9 463 //向进程发送SIGKILL信号，终止进程

localhost:~$
```

（多次注册同一信号会调用哪一个handler？最后安装的handler）

4.2 sigaction()（signal action）

```
#include <signal.h>
#include <sigaction.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

的对象用来保存返回的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信号等等。

`sigaction`结构定义如下：

```
struct sigaction {
    union{
        __sighandler_t sa_handler;
        void (*sa_sigaction)(int,siginfo_t *, void *);
    }_u
    //……
    __sigset_t sa_mask;
    //……
    unsigned long sa_flags;
}
```

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_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，第二个参数是指向`siginfo_t`结构体的指针。`siginfo_t`结构体中包含信号携带的数据值，其部分源码如下（siginfo_t.h）：

```
#define si_pid      _sifields._kill.si_pid
#define si_value    _sifields._rt.si_sigval
#define si_int      _sifields._rt.si_sigval.sival_int
#define si_ptr      _sifields._rt.si_sigval.sival_ptr
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*/
        int _pad[SI_PAD_SIZE];//确保分配足够大的存储空间
        struct{//对SIGKILL有意义的结构
            ...
        }_kill;
        ... ...
        ... ...
        //对SIGILL, SIGFPE, SIGSEGV, SIGBUS有意义的结构
        /* POSIX.lb signals=基于POSIX发送的信号*/
        struct{
            /* Sending process ID.  */
            __pid_t si_pid;
            /* Real user ID of sending process.  */
```

```
            __uid_t si_uid;
            /* Signal value.  */
            __sigval_t si_sigval;
        }_rt;
        ... ...
    }_sifield
}
```

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`sigval`中的数据就将拷贝到信号处理函数的第二个参数`siginfo_t`中。用户进程发送的信号遵循POSIX协议，故信号包含的信息会被存储到`_sifields._rt.si_sigval`联合体下，通过宏定义，我们可以直接使用`si_value`,`si_int`,`si_ptr`来获取信号包含的信息。
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。

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：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）。

换言之，如果设置标志位为`SA_SIGINFO`表明接受的信号含有信息，我们使用`sa_sigaction`指向的信号处理函数来处理，否则，交给`sa_handler`处理。

5 信号的发送

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

5.1 kill()

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid,int signo)
```

该系统调用可以用来向任何进程或进程组发送任何信号。参数`pid`的值为信号的接收进程

> *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。该调用执行成功时，返回值为0；错误时，返回-1，并设置相应的错误代码errno。下面是一些可能返回的错误代码：

> *EINVAL：指定的信号sig无效。*
>
> *ESRCH：参数pid指定的进程或进程组不存在。注意，在进程表项中存在的进程，可能是一个还没有被wait收回，但已经终止执行的僵死进程。*
>
> *EPERM： 进程没有权力将这个信号发送到指定接收信号的进程。因为，一个进程被允许将信号发送到进程pid时，必须拥有root权力，或者是发出调用的进程的UID 或EUID与指定接收的进程的UID或保存用户ID（savedset-user-ID）相同。如果参数pid小于-1，即该信号发送给一个组，则该错误表示组中有成员进程不能接收该信号。*

5.2 sigqueue（）（signal queue）

```
#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对应`sigaction()`注册的3参数信号处理函数`sa_sigaction`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。

5.3 alarm（）

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds)
```

系统调用`alarm`安排内核为调用进程在指定的seconds秒后发出一个`SIGALRM`的信号。如果指定的参数seconds为0，则不再发送`SIGALRM`信号。后一次设定将取消前一次的设定。该调用返回值为上次定时调用到发送之间剩余的时间，或者因为没有前一次定时调用而返回0。

注意，在使用时，alarm只设定为发送一次信号，如果要多次发送，就要多次使用alarm调用。

5.4 setitimer()（set inter timer）

现在的系统中很多程序不再使用`alarm`调用，而是使用`setitimer`调用来设置定时器，用`getitimer`来得到定时器的状态，这两个调用的声明格式如下：

```
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value,
struct itimerval *ovalue);
```

在使用这两个调用的进程中加入以下头文件：

```
#include <sys/time.h>
```

该系统调用给进程提供了三个定时器，它们各自有其独有的计时域，当其中任何一个到达，就发送一个相应的信号给进程，并使得计时器重新开始。三个计时器由参数which指定，如下所示：

1. `TIMER_REAL`：按实际时间计时，计时到达将给进程发送`SIGALRM`信号。
2. `ITIMER_VIRTUAL`：仅当进程执行时才进行计时。计时到达将发送`SIGVTALRM`信号给进程。
3. `ITIMER_PROF`：当进程执行时和系统为该进程执行动作时都计时。与`ITIMER_VIR-TUAL`是一对，该定时器经常用来统计进程在用户态和内核态花费的时间。计时到达将发送`SIGPROF`信号给进程。

定时器中的参数value用来指明定时器的时间，其结构如下：

```
struct itimerval {
    struct timeval it_interval; /* 下一次的取值 */
    struct timeval it_value; /* 本次的设定值 */
};
```

该结构中`timeval`结构定义如下：

```
struct timeval {
    long tv_sec; /* 秒 */
    long tv_usec; /* 微秒，1秒 = 1000000 微秒*/
};
```

在`setitimer`调用中，参数`ovalue`如果不为空，则其中保留的是上次调用设定的值。定时器将`it_value`递减到0时，产生一个信号，并将`it_value`的值设定为`it_interval`的值，然后重新开始计时，如此往复。当`it_value`设定为0时，计时器停止，或者当它计时到期，而`it_interval`为0时停止。调用成功时，返回0；错误时，返回-1，并设置相应的错误代码errno：

> *EFAULT：参数value或ovalue是无效的指针。*
>
> *EINVAL：参数which不是ITIMER_REAL、ITIMER_VIRT或ITIMER_PROF中的一个。*

下面是关于`setitimer`调用的一个简单示范，在该例子中，每隔一秒发出一个`SIGALRM`，每隔0.5秒发出一个`SIGVTALRM`信号：

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/time.h>

int sec;

void sigroutine(int signo) {
    switch (signo) {
    case SIGALRM:
        printf("Catch a signal -- SIGALRM ");
        break;
    case SIGVTALRM:
        printf("Catch a signal -- SIGVTALRM ");
        break;
    }
    return;
}

int main()
{
    struct itimerval value,value2,ovalue;
    sec = 5;
    printf("process id is %d ",getpid());

```

```
    signal(SIGALRM, sigroutine);
    signal(SIGVTALRM, sigroutine);

    value.it_value.tv_sec = 1;
    value.it_value.tv_usec = 0;
    value.it_interval.tv_sec = 1;
    value.it_interval.tv_usec = 0;

    seti1timer(ITIMER_REAL, &value, &ovalue);

    value2.it_value.tv_sec = 0;
    value2.it_value.tv_usec = 500000;
    value2.it_interval.tv_sec = 0;
    value2.it_interval.tv_usec = 500000;

    setitimer(ITIMER_VIRTUAL, &value2, &ovalue);
    for (;;) ;
}
```

该例子的屏幕拷贝如下：

```
localhost:~$ ./timer_test

process id is 579

Catch a signal - SIGVTALRM

Catch a signal - SIGALRM

Catch a signal - SIGVTALRM

Catch a signal - SIGVTALRM

Catch a signal - SIGALRM

Catch a signal -GVTALRM
```

5.5 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。

5.6 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。

6 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，每个信号占用一位。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>
/*Signal empty set*/
int sigemptyset(sigset_t *set);
/*Signal fill set*/
int sigfillset(sigset_t *set);
/*Signal add set*/
int sigaddset(sigset_t *set, int signum)
/*Signal delete set*/
int sigdelset(sigset_t *set, int signum);
/*Signal is member*/
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。
```

7 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：

```c
#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));
```

7.1 sigprocmask()（signal process 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：

码	含义
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.

7.2 sigpending()（signal pending）

`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。

7.3 sigsuspend()（signal suspend）

`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。

8 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。

8.1 信号发送与处理

实现一个信号接收程序`sigreceive`（其中信号安装由`sigaction()`）。

```
```

```c
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.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);
}
```

`atoi()`函数[2]
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。

8.2 信号传递附加信息

主要包括两个实例：
向进程本身发送信号，并传递指针参数

```c
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void new_op(int,siginfo_t*,void*);
```

```c
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);//置空未决信号集
    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，允许传说参数信息给new_op
    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。

8.3 不同进程间传递整型参数：

把8.1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：

```c
#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。

```c
#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：8.3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

8.4 信号阻塞及信号集操作

```
#include "signal.h"
#include "unistd.h"
static void my_op(int);
void 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。

---

1. 进程（程序）开始运行时，由Linux系统调用自己的系统函数，在内存中开辟task_struct结构体，又叫进程表。这个结构体的成员项非常多，多达近300个。task_struct结构体专门用于存放进程在运行过程中，所涉及到的所有与进程相关的信息。其中，文件描述符表就被包含在了task_struct结构体当中。在进程运行结束后，进程表所占用的内存空间，会被释放。 ↩

2. 函数原型：int atoi(const char *nptr);（ASCII to Integer）是把字符串转换成整型数的一个函数。atoi函数会扫描参数 nptr字符串，会跳过前面的空白字符（例如空格，tab缩进）等。如果 nptr不能转换成 int 或者 nptr为空字

符串，那么将返回 0 。特别注意，该函数要求被转换的字符串是按十进制数理解的。atoi输入的字符串对应数字存在大小限制（与int类型大小有关），若其过大可能报错-1。 ↩