[linux系统编程之信号（一）：中断与信号](http://www.cnblogs.com/mickole/p/3189156.html)

**一，什么是中断？**

**1．中断的基本概念**

中断是指计算机在执行期间，系统内发生任何非寻常的或非预期的急需处理事件，使得CPU暂时中断当前正在执行的程序而转去执行相应的事件处理程序，待处理完毕后又返回原来被中断处继续执行或调度新的进程执行的过程。引起中断发生的事件被称为中断源。中断源向CPU发出的请求中断处理信号称为中断请求，而CPU收到中断请求后转到相应的事件处理程序称为中断响应。

在有些情况下，尽管产生了中断源和发出了中断请求，但CPU内部的处理器状态字PSW的中断允许位已被清除，从而不允许CPU响应中断。这种情况称为禁止中断。CPU禁止中断后只有等到PSW的中断允许位被重新设置后才能接收中断。禁止中断也称为关中断，PSW的中断允许位的设置也被称为开中断。开中断和关中断是为了保证某段程序执行的原子性。

还有一个比较常用的概念是中断屏蔽。中断屏蔽是指在中断请求产生之后，系统有选择地封锁一部分中断而允许另一部分中断仍能得到响应。不过，有些中断请求是不能屏蔽甚至不能禁止的，也就是说，这些中断具有最高优先级，只要这些中断请求一旦提出，CPU必须立即响应。例如，电源掉电事件所引起的中断就是不可禁止和不可屏蔽的。

**2．中断的分类与优先级**

根据系统对中断处理的需要，操作系统一般对中断进行分类并对不同的中断赋予不同的处理优先级，以便在不同的中断同时发生时，按轻重缓急进行处理。

根据中断源产生的条件，可把中断分为外中断和内中断。外中断是指来自处理器和内存外部的中断，包括I／0设备发出的I／O中断、外部信号中断(例如用户键人ESC键)。各种定时器引起的时钟中断以及调试程序中设置的断点等引起的调试中断等。外中断在狭义上一般被称为中断。

内中断主要指在处理器和内存内部产生的中断。内中断一般称为陷阱(trap)或异常。它包括程序运算引起的各种错误，如地址非法、校验错、页面失效、存取访问控制错、算术操作溢出、数据格式非法、除数为零、非法指令、用户程序执行特权指令、分时系统中的时间片中断以及从用户态到核心态的切换等都是陷阱的例子。

为了按中断源的轻重缓急处理响应中断，操作系统为不同的中断赋予不同的优先级。例如在UNIX系统中，外中断和陷阱的优先级共分为8级。为了禁止中断或屏蔽中断，CPU的处理器状态字PSW中也设有相应的优先级。如果中断源的优先级高于PSW的优先级，则CPU响应该中断源的请求；反之，CPU屏蔽该中断源的中断请求。

各中断源的优先级在系统设计时给定，在系统运行时是固定的。而处理器的优先级则根据执行情况由系统程序动态设定。

除了在优先级的设置方面有区别之外，中断和陷阱还有如下主要区别：

陷阱通常由处理器正在执行的现行指令引起，而中断则是由与现行指令无关的中断源引起的。陷阱处理程序提供的服务为当前进程所用，而中断处理程序提供的服务则不是为了当前进程的。

CPU执行完一条指令之后，下一条指令开始之前响应中断，而在一条指令执行中也可以响应陷阱。例如执行指令非法时，尽管被执行的非法指令不能执行结束，但CPU仍可对其进行处理。

**3．软中断**

软中断的概念主要来源于UNIX系统。软中断是对应于硬中断而言的。通过硬件产生相应的中断请求，称为硬中断。而软中断则不然，它是在通信进程之间通过模拟硬中断而实现的一种通信方式。中断源发出软中断信号后，CPU或者接收进程在“适当的时机”进行中断处理或者完成软中断信号所对应的功能。这里“适当的时机”，表示接收软中断信号的进程须等到该接收进程得到处理器之后才能进行。如果该接收进程是占据处理器的，那么，该接收进程在接收到软中断信号后将立即转去执行该软中断信号所对应的功能。

**4．中断处理过程**

一旦CPU响应中断，转人中断处理程序，系统就开始进行中断处理。下面对中断处理过程进行详细说明：

1)CPU检查响应中断的条件是否满足。CPU响应中断的条件是：有来自于中断源的中断请求、CPU允许中断。如果中断响应条件不满足，则中断处理无法进行。

2)如果CPU响应中断，则CPU关中断，使其进入不可再次响应中断的状态。

3)保存被中断进程现场。为了在中断处理结束后能使进程正确地返回到中断点，系统必须保存当前处理器状态字PSW和程序计数器PC等的值。这些值一般保存在特定堆栈或硬件寄存器中。

4)分析中断原因，调用中断处理子程序。在多个中断请求同时发生时，处理优先级最高的中断源发出的中断请求。在系统中，为了处理上的方便，通常都是针对不同的中断源编制有不同的中断处理子程序(陷阱处理子程序)。这些子程序的人口地址(或陷阱指令的人口地址)存放在内存的特定单元中。再者，不同的中断源也对应着不同的处理器状态字PSW。这些不同的PSW被放在相应的内存单元中，与中断处理子程序人口地址一起构成中断向量。显然，根据中断或陷阱的种类，系统可由中断向量表迅速地找到该中断响应的优先级、中断处理子程序(或陷阱指令)的入口地址和对应的PSW。

5)执行中断处理子程序。对陷阱来说，在有些系统中则是通过陷阱指令向当前执行进程发出软中断信号后调用对应的处理子程序执行。

6)退出中断，恢复被中断进程的现场或调度新进程占据处理器。

7)开中断，CPU继续执行。

**5．设备管理程序与中断方式**

处理器的高速和输入输出设备低速之间的矛盾，是设备管理要解决的一个重要问题。为了提高整体效率，减少在程序直接控制方式中的CPU等待时间以及提高系统的并行工作效率，采用中断方式来控制输入输出设备和内存与CPU之间的数据传送，是很有必要的。

在硬件结构上，这种方式要求CPU与输入输出设备(或控制器)之间有相应的中断请求线，而且在输入输出设备控制器的控制状态寄存器上有相应的中断允许位。

**二，什么是信号？**

**1、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。

**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：（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。

（一）、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。

（二）、实时信号与非实时信号

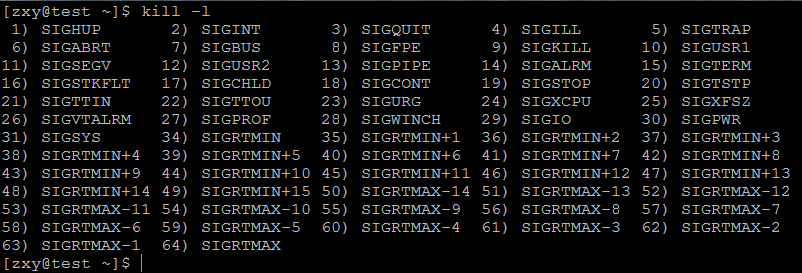
早期Unix系统只定义了32种信号，Ret hat7.2支持64种信号，编号0-63(SIGRTMIN=31，SIGRTMAX=63)，将来可能进一步增加，这需要得到内核的支持。前32种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的CTRL ^C时，会产生SIGINT信号，对该信号的默认反应就是进程终止。后32个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收。实时信号是POSIX标准的一部分，可用于应用进程。

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。

（三）,信号名称

查看linux所支持的信号可用：kill –l

共64种：

[](http://images.cnitblog.com/blog/529981/201307/14104400-0541f2d712e04ca793a4dbeecf902d1e.png)

[](http://images.cnitblog.com/blog/529981/201307/14104401-e14c06f2ab494f408848ce388450a2df.png)

[](http://images.cnitblog.com/blog/529981/201307/14104402-436280b09f8d48f689cbbf003f8ade82.png)

**3****、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。

注：本文参考：<http://blog.csdn.net/lmh12506/article/details/6681663>和

<http://blog.csdn.net/johnny710vip/article/details/6990514>

# [linux系统编程之信号（二）：信号处理流程（产生、注册、注销、执行）](http://www.cnblogs.com/mickole/p/3189666.html)

    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：

* 信号诞生
* 信号在进程中注册
* 信号在进程中的注销
* 信号处理函数执行

#### 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

A 缺省的动作是终止进程

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

#### 2 信号在目标进程中注册

在进程表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。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，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 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）。

# [linux系统编程之信号（三）：信号安装、signal、kill，arise讲解](http://www.cnblogs.com/mickole/p/3190783.html)

## 一，信号安装

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。

## 二，signal()用法

#include <signal.h>

typedef void (\*\_\_sighandler\_t) (int);

#define SIG\_ERR ((\_\_sighandler\_t) -1)

#define SIG\_DFL ((\_\_sighandler\_t) 0)

#define SIG\_IGN ((\_\_sighandler\_t) 1)

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，返回最后一次也就是上一次为安装信号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。

man帮助说明：

DESCRIPTION   
       The behavior of signal() varies across Unix versions, and has also var-   
       ied 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  func-   
       tion (a "signal handler").

       If  the signal signum is delivered to the process, then one of the fol-   
       lowing happens:

       \*  If the disposition is set to SIG\_IGN, then the signal is ignored.

       \*  If the disposition is set to SIG\_DFL, then the default action  asso-   
          ciated with the signal (see signal(7)) occurs.

       \*  If  the disposition is set to a function, then first either the dis-   
          position is reset to SIG\_DFL, or the signal is blocked  (see  Porta-   
          bility  below), and then handler is called with argument signum.  If   
          invocation of the handler caused the signal to be blocked, then  the

          signal is unblocked upon return from the handler.

       The signals SIGKILL and SIGSTOP cannot be caught or ignored.

RETURN VALUE   
       signal()  returns  the previous value of the signal handler, or SIG\_ERR   
       on error.

示例程序：

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

void sig\_handler(int signo);

int main(void)

{

printf("mian is waiting for a signal\n");

if(signal(SIGINT,sig\_handler) == SIG\_ERR){

perror("signal errror");

exit(EXIT\_FAILURE);

}

for(; ;);//有时间让我们发送信号

return 0;

}

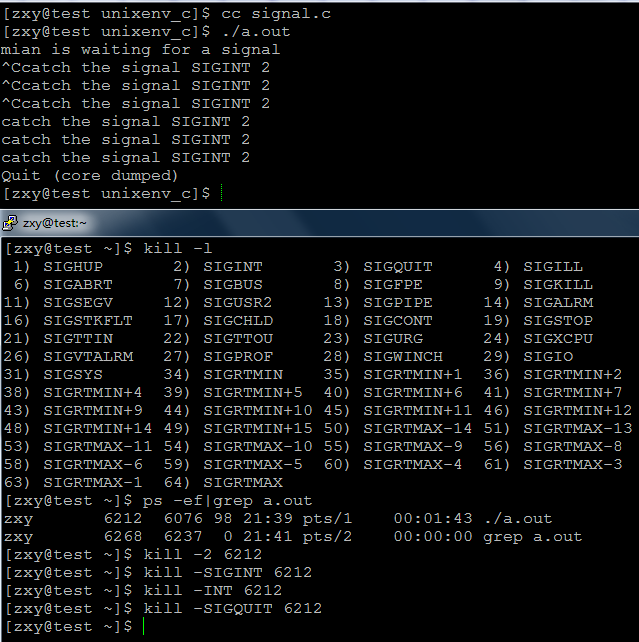
void sig\_handler(int signo)

{

printf("catch the signal SIGINT %d\n",signo);

}

结果：

[](http://images.cnitblog.com/blog/529981/201307/15105413-f3c8a980a41c4e51880491943ba80a6f.png)

可知我们捕获了SIGINT信号，每当我们按下ctrl+c或利用kill发送SIGINT信号时，执行我们安装的信号处理函数，当我们按下：ctrl+\或kill –SIGQUIT pid发送SIGQUIT信号时，程序退出，那是因为进程对SIGQUIT信号的默认处理动作是退出程序。

现在我们来获得进程的最后一次为安装信号时所指定的处理函数：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

#include <sys/types.h>

void sig\_handler(int signo);

int main(void)

{

printf("main is waiting for a signal\n");

\_\_sighandler\_t prehandler;

prehandler = signal(SIGINT,sig\_handler);

if(prehandler == SIG\_ERR){

perror("signal errror");

exit(EXIT\_FAILURE);

}

printf("the previous value of the signal handler is %d\n",(int)prehandler);

//for(; ;);//有时间让我们发送信号

return 0;

}

void sig\_handler(int signo)

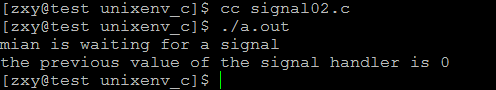
{

printf("catch the signal SIGINT %d\n",signo);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15105415-7ff46d12280640729a5301907c666b99.png)

为0，由前面的宏定义：#define SIG\_DFL ((\_\_sighandler\_t) 0)，可知处理动作为SIG\_DFL，而SIGINT默认的处理动作就是终止进程

示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

#include <sys/types.h>

void sig\_handler(int signo);

int main(void)

{

printf("main is waiting for a signal\n");

\_\_sighandler\_t prehandler;

prehandler = signal(SIGINT,SIG\_DFL);

if(prehandler == SIG\_ERR){

perror("signal errror");

exit(EXIT\_FAILURE);

}

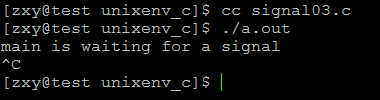
for(; ;);//有时间让我们发送信号

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15105415-618283b114ac437fa311cf4f12a3b772.png)

当按下ctrl+c时发送SIGINT信号给进程，然后进程终止

## 三，kill()发送信号

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

这里我们先将kill函数使用：

#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 除发送给每一个调用进程有权限发送的进程除自身及1（init）进程外

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。

man帮助说明：

DESCRIPTION   
       The  kill()  system  call can be used to send any signal to any process   
       group or process.

       If pid is positive, then signal sig is sent to the process with the  ID   
       specified by pid.

       If pid equals 0, then sig is sent to every process in the process group   
       of the calling process.

       If pid equals -1, then sig is sent to every process for which the call-   
       ing  process  has  permission  to  send  signals,  except for process 1   
       (init), but see below.

       If pid is less than -1, then sig is sent to every process in  the  pro-   
       cess group whose ID is -pid.

       If  sig  is 0, then no signal is sent, but error checking is still per-   
       formed; this can be used to check for the existence of a process ID  or   
       process group ID.

       For  a  process  to  have permission to send a signal it must either be   
       privileged (under Linux: have the CAP\_KILL capability), or the real  or   
       effective  user  ID of the sending process must equal the real or saved   
       set-user-ID of the target process.  In the case of SIGCONT it  suffices   
       when the sending and receiving processes belong to the same session.

RETURN VALUE   
       On success (at least one signal was sent), zero is returned.  On error,   
       -1 is returned, and errno is set appropriately.

ERRORS   
       EINVAL An invalid signal was specified.

       EPERM  The process does not have permission to send the signal  to  any   
              of the target processes.

       ESRCH  The  pid or process group does not exist.  Note that an existing   
              process might be a zombie, a  process  which  already  committed   
              termination, but has not yet been wait(2)ed for.

示例程序：

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig);

int main(int argc, char \*argv[])

{

if (signal(SIGUSR1, handler) == SIG\_ERR)

ERR\_EXIT("signal error");

pid\_t pid = fork();

if (pid == -1)

ERR\_EXIT("fork error");

if (pid == 0)

{

sleep(1);

kill(getppid(), SIGUSR1);

exit(EXIT\_SUCCESS);

}

int n = 5;

do

{

printf("the number of seconds left to sleep is %d s\n",n);

n = sleep(n);

} while (n > 0);

return 0;

}

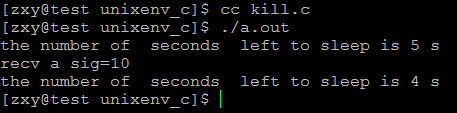
void handler(int sig)

{

printf("recv a sig=%d\n", sig);

}

结果：

[](http://images.cnitblog.com/blog/529981/201307/15105416-dc532a47abd04cfe9aec479acb789151.png)   
以上程序里有子进程给父进程发送SIGUSR1信号，父进程收到信号后，睡眠被中断，然后去执行信号处理函数，返回后继续睡眠剩余的时间后退出程序。

现在利用kill给与给定pid同组所有进程发送信号：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig);

int main(int argc, char \*argv[])

{

if (signal(SIGUSR1, handler) == SIG\_ERR)

ERR\_EXIT("signal error");

pid\_t pid = fork();

if (pid == -1)

ERR\_EXIT("fork error");

if (pid == 0)

{

pid = getpgrp();

kill(-pid, SIGUSR1);

//kilpg(getpgrp(), SIGUSR1);

exit(EXIT\_SUCCESS);

}

int n = 5;

do

{

n = sleep(n);

} while (n > 0);

return 0;

}

void handler(int sig)

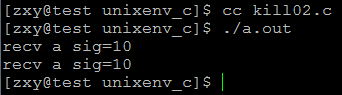
{

printf("recv a sig=%d\n", sig);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15105417-702a371ec3a146cf98718fa2a584b10a.png)

可知收到进行了两次信号处理函数的执行：因为当前所属组中只有父子两个进程，从上可知有两种方式给组进程发送信号：kill和killpg

## 四，arise函数

#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。

man帮助说明：

DESCRIPTION   
       The  raise()  function sends a signal to the calling process or thread.   
       In a single-threaded program it is equivalent to

           kill(getpid(), sig);

       In a multithreaded program it is equivalent to

           pthread\_kill(pthread\_self(), sig);

       If the signal causes a handler to be called, raise() will  only  return   
       after the signal handler has returned.

RETURN VALUE   
       raise() returns 0 on success, and non-zero for failure.

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

void sig\_handler(int signo);

int main(void)

{

printf("mian is waiting for a signal\n");

if(signal(SIGINT,sig\_handler) == SIG\_ERR){

perror("signal errror");

exit(EXIT\_FAILURE);

}

printf("useing raise to send a signal to himself\n");

raise(SIGINT);

sleep(1);

printf("useing kill to send a signal to himself\n");

kill(getpid(),SIGINT);

return 0;

}

void sig\_handler(int signo)

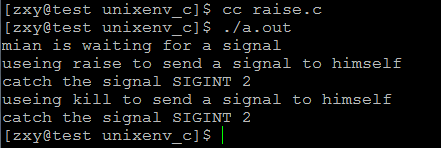
{

printf("catch the signal SIGINT %d\n",signo);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15105417-5ddf900b25d541a8b6837cdf6ef31366.png)

可知两种方式都可以给自身发送信号。

# [linux系统编程之信号（四）：alarm和可重入函数](http://www.cnblogs.com/mickole/p/3190845.html)

## 一，alarm()

在将可重入函数之前我们先来了解下alarm()函数使用：

#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。

man帮助说明：

DESCRIPTION   
       alarm()  arranges  for  a SIGALRM signal to be delivered to the calling   
       process in seconds seconds.

       If seconds is zero, no new alarm() is scheduled.

       In any event any previously set alarm() is canceled.

RETURN VALUE   
       alarm() returns the number of seconds remaining  until  any  previously   
       scheduled alarm was due to be delivered, or zero if there was no previ-   
       ously scheduled alarm.

示例：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig);

int main(int argc, char \*argv[])

{

if (signal(SIGALRM, handler) == SIG\_ERR)

ERR\_EXIT("signal error");

alarm(1);

for (;;)

pause();

return 0;

}

void handler(int sig)

{

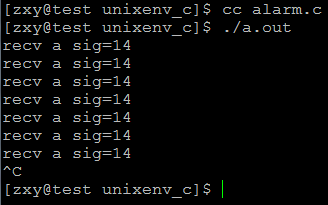
printf("recv a sig=%d\n", sig);

alarm(1);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15112934-3ad9f42d5489492fbf1e6e26b50870fe.png)

因为在使用时，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，所以可在信号处理函数中调用alarm()实现每隔指点秒受发送SIGALRM信号。

## 二，可重入函数

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。

示例程序：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

typedef struct

{

int a;

int b;

} TEST;

TEST g\_data;

void handler(int sig);

int main(int argc, char \*argv[])

{

TEST zeros = {0, 0};

TEST ones = {1, 1};

if (signal(SIGALRM, handler) == SIG\_ERR)

ERR\_EXIT("signal error");

g\_data = zeros;

alarm(1);

for (;;)

{

g\_data = zeros;

g\_data = ones;

}

return 0;

}

void unsafe\_fun()

{

printf("%d %d\n", g\_data.a, g\_data.b);

}

void handler(int sig)

{

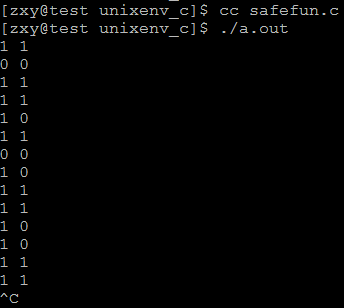
unsafe\_fun();

alarm(1);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15112935-82df6bdec112471a821026c43eb1319d.png)

也是程序创建了一个结构体，设置一个全局变量，然后在main函数中利用两个局部变量分别给全局变量赋值，由于这个赋值操作是可被中断的，如以上每一次结构体的赋值可视为两步：

g\_data.a=zeros.a;

g\_data.b=zeros.b;

所以当g\_data.a=one.a;做完然后被中断，跑去执行处理函数，在处理函数中调用unsafe\_fun（）打印全局变量值，可知结果是全局变量a值变了，b值还是之前的没来的及改变，所以出现了1,0

所以结果不确定

# [linux系统编程之信号（五）：信号集操作函数，信号阻塞与未决](http://www.cnblogs.com/mickole/p/3191281.html)

### 一，信号集及相关操作函数

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（64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>

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，相当于64为置0；

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，相当于64为都置1；

sigaddset(sigset\_t \*set, int signum)在set指向的信号集中加入signum信号，相当于将给定信号所对应的位置1；

sigdelset(sigset\_t \*set, int signum)在set指向的信号集中删除signum信号，相当于将给定信号所对应的位置0；

sigismember(const sigset\_t \*set, int signum)判定信号signum是否在set指向的信号集中，相当于检查给定信号所对应的位是0还是1。

示例程序：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <signal.h>

#include <sys/types.h>

void print\_sigset(sigset\_t \*set);

int main(void)

{

sigset\_t myset;

sigemptyset(&myset);

sigaddset(&myset,SIGINT);

sigaddset(&myset,SIGQUIT);

sigaddset(&myset,SIGUSR1);

sigaddset(&myset,SIGRTMIN);

print\_sigset(&myset);

return 0;

}

void print\_sigset(sigset\_t \*set)

{

int i;

for(i = 1; i < NSIG; ++i){

if(sigismember(set,i))

printf("1");

else

printf("0");

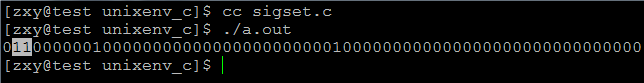
}

putchar('\n');

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15154938-c0af52d77dc9484593b48548050ff276.png)

可以看到添加信号的相应位置1.

### 二，信号阻塞与未决

man帮助说明：

##### Signal mask and pending signals

A signal may be *blocked*, which means that it will not be delivered

until it is later unblocked. Between the time when it is generated

and when it is delivered a signal is said to be *pending*.

Each thread in a process has an independent *signal mask*, which

indicates the set of signals that the thread is currently blocking.

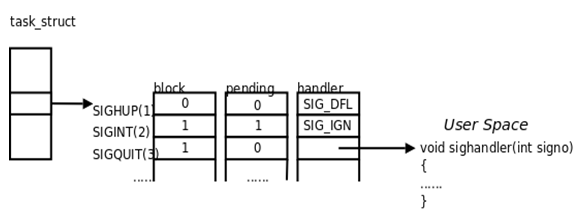
A thread can manipulate its signal mask using [pthread\_sigmask(3)](http://man7.org/linux/man-pages/man3/pthread_sigmask.3.html). In

a traditional single-threaded application, [sigprocmask(2)](http://man7.org/linux/man-pages/man2/sigprocmask.2.html) can be used

to manipulate the signal mask.

执行信号的处理动作称为信号递达（Delivery），信号从产生到递达之间的状态，称为信号未决（Pending）。进程可以选择阻塞（Block）某个信号。被阻塞的信号产生时将保持在未决状态，直到进程解除对此信号的阻塞，才执行递达的动作。注意，阻塞和忽略是不同的，只要信号被阻塞就不会递达，而忽略是在递达之后可选的一种处理动作。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。

信号在内核中的表示可以看作是这样的：

[](http://images.cnitblog.com/blog/529981/201307/15154939-e66d8b4992824bd28fed4d6f093b6b14.png)

看图说话：

block集（阻塞集、屏蔽集）：一个进程所要屏蔽的信号，在对应要屏蔽的信号位置1

pending集（未决信号集）：如果某个信号在进程的阻塞集中，则也在未决集中对应位置1，表示该信号不能被递达，不会被处理

handler（信号处理函数集）：表示每个信号所对应的信号处理函数，当信号不在未决集中时，将被调用

以下是与信号阻塞及未决相关的函数操作：

#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：

* 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，相当于：mask=mask|set
* 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，相当于：mask=mask|~set
* SIG\_SETMASK 更新进程阻塞信号集为set指向的信号集，相当于mask=set

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。

示例程序：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig);

void printsigset(sigset\_t \*set)

{

int i;

for (i=1; i<NSIG; ++i)

{

if (sigismember(set, i))

putchar('1');

else

putchar('0');

}

printf("\n");

}

int main(int argc, char \*argv[])

{

sigset\_t pset;

sigset\_t bset;

sigemptyset(&bset);

sigaddset(&bset, SIGINT);

if (signal(SIGINT, handler) == SIG\_ERR)

ERR\_EXIT("signal error");

if (signal(SIGQUIT, handler) == SIG\_ERR)

ERR\_EXIT("signal error");

sigprocmask(SIG\_BLOCK, &bset, NULL);//将信号加入进程阻塞集中

for (;;)

{

sigpending(&pset);

printsigset(&pset);

sleep(1);

}

return 0;

}

void handler(int sig)

{

if (sig == SIGINT)

printf("recv a sig=%d\n", sig);

else if (sig == SIGQUIT)

{

sigset\_t uset;

sigemptyset(&uset);

sigaddset(&uset, SIGINT);

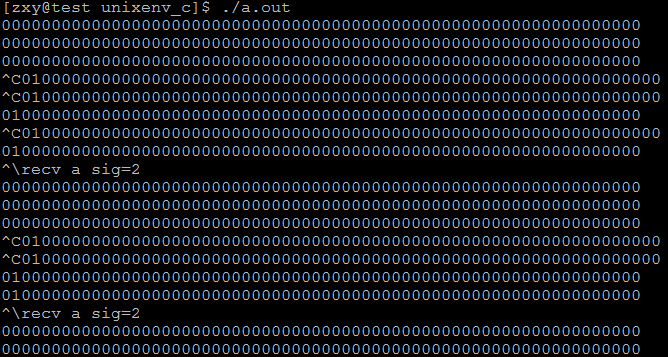
sigprocmask(SIG\_UNBLOCK, &uset, NULL);

}

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15154940-79af074dae33439c82eae751c404a3e4.png)

    说明：程序首先将SIGINT信号加入进程阻塞集（屏蔽集）中，一开始并没有发送SIGINT信号，所以进程未决集中没有处于未决态的信号，当我们连续按下ctrl+c时，向进程发送SIGINT信号，由于SIGINT信号处于进程的阻塞集中，所以发送的SIGINT信号不能递达，也是就是处于未决状态，所以当我打印未决集合时发现SIGINT所对应的位为1，现在我们按下ctrl+\，发送SIGQUIT信号，由于此信号并没被进程阻塞，所以SIGQUIT信号直接递达，执行对应的处理函数，在该处理函数中解除进程对SIGINT信号的阻塞，所以之前发送的SIGINT信号递达了，执行对应的处理函数，但由于SIGINT信号是不可靠信号，不支持排队，所以最终只有一个信号递达。

[复制代码](javascript:void(0);)

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <error.h>

#include <string.h>

#include <unistd.h>

/\* 版本1, 可靠信号将被递送多次 \*/

#define MYSIGNAL SIGRTMIN+5

/\* 版本2, 不可靠信号只被递送一次 \*/

//#define MYSIGNAL SIGTERM

void sig\_handler(int signum)

{

psignal(signum, "catch a signal");

}

int main(int argc, char \*\*argv)

{

sigset\_t block, pending;

int sig, flag;

/\* 设置信号的handler \*/

signal(MYSIGNAL, sig\_handler);

/\* 屏蔽此信号 \*/

sigemptyset(&block);

sigaddset(&block, MYSIGNAL);

printf("block signal\n");

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

/\* 发两次信号, 看信号将会被触发多少次 \*/

printf("---> send a signal --->\n");

kill(getpid(), MYSIGNAL);

printf("---> send a signal --->\n");

kill(getpid(), MYSIGNAL);

/\* 检查当前的未决信号 \*/

flag = 0;

sigpending(&pending);

for (sig = 1; sig < NSIG; sig++) {

if (sigismember(&pending, sig)) {

flag = 1;

psignal(sig, "this signal is pending");

}

}

if (flag == 0) {

printf("no pending signal\n");

}

/\* 解除此信号的屏蔽, 未决信号将被递送 \*/

printf("unblock signal\n");

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

/\* 再次检查未决信号 \*/

flag = 0;

sigpending(&pending);

for (sig = 1; sig < NSIG; sig++) {

if (sigismember(&pending, sig)) {

flag = 1;

psignal(sig, "a pending signal");

}

}

if (flag == 0) {

printf("no pending signal\n");

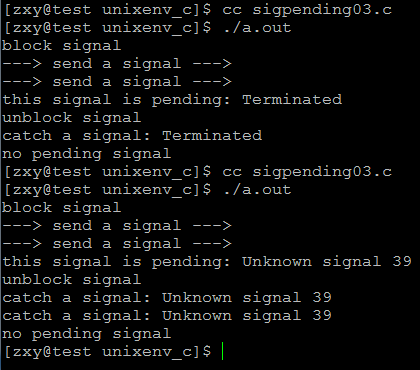
}

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15154941-42d76476d711422eb4d12a94a7c26b1b.png)

两次执行结果不同：第一次连续发送两次不可靠信号，最后解除阻塞时，只有一个递达，说明不可靠信号不支持排队。

第二次执行时，连续两次发送可靠信号，解除阻塞后，都递达，说明可靠信号支持排队。

ok，这节就写到这吧

# [linux系统编程之信号（六）：信号发送函数sigqueue和信号安装函数sigaction](http://www.cnblogs.com/mickole/p/3191804.html)

### 一，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：

[复制代码](javascript:void(0);)

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)；

}

[复制代码](javascript:void(0);)

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：

[复制代码](javascript:void(0);)

typedef struct siginfo\_t{

int si\_signo;//信号编号

int si\_errno;//如果为非零值则错误代码与之关联

int si\_code;//说明进程如何接收信号以及从何处收到

pid\_t si\_pid;//适用于SIGCHLD，代表被终止进程的PID

pid\_t si\_uid;//适用于SIGCHLD,代表被终止进程所拥有进程的UID

int si\_status;//适用于SIGCHLD，代表被终止进程的状态

clock\_t si\_utime;//适用于SIGCHLD，代表被终止进程所消耗的用户时间

clock\_t si\_stime;//适用于SIGCHLD，代表被终止进程所消耗系统的时间

sigval\_t si\_value;

int si\_int;

void \* si\_ptr;

void\* si\_addr;

int si\_band;

int si\_fd;

};

[复制代码](javascript:void(0);)

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

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。

sigval结构体：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。

siginfo\_t.si\_value与sigqueue(pid\_t pid, int sig, const union sigval val)第三个参数关联即：

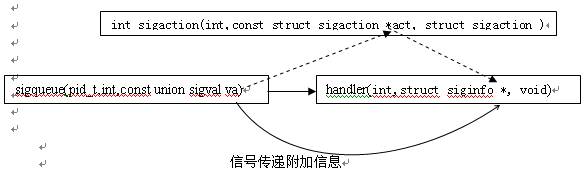
所以通过siginfo\_t.si\_value可以获得sigqueue(pid\_t pid, int sig, const union sigval val)第三个参数传递过来的数据。

如：siginfo\_t.si\_value.sival\_int或siginfo\_t.si\_value.sival\_ptr

其实siginfo\_t.si\_int直接与sigval.sival\_int关联

siginfo\_t.si\_ptr直接与sigval.sival\_ptr关联，所以也可同这种方式获得sigqueue发送过来的数据。

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，处理程序执行完后，被阻塞的信号开始执行。

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。

### 二，sigqueue()

之前学过kill,raise,alarm,abort等功能稍简单的信号发送函数，现在我们学习一种新的功能比较强大的信号发送函数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对应sig 注册的3参数信号处理函数的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。

### 三，sigqueue与sigaction应用实例

实例一：利用sigaction安装SIGINT信号

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig);

int main(int argc, char \*argv[])

{

struct sigaction act;

act.sa\_handler = handler;

sigemptyset(&act.sa\_mask);

act.sa\_flags = 0;

//因为不关心SIGINT上一次的struct sigaction所以，oact为NULL

//与signal(handler,SIGINT)相同

if (sigaction(SIGINT, &act, NULL) < 0)

ERR\_EXIT("sigaction error\n");

for (;;)

pause();

return 0;

}

void handler(int sig)

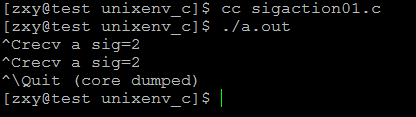
{

printf("recv a sig=%d\n", sig);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15192103-bc05cc4fdf954d08a22054b808477a50.png)

实例二：利用sigaction实现signal，实际上signal底层实现就是利用sigaction

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig);

\_\_sighandler\_t my\_signal(int sig, \_\_sighandler\_t handler);

int main(int argc, char \*argv[])

{

my\_signal(SIGINT, handler);

for (;;)

pause();

return 0;

}

\_\_sighandler\_t my\_signal(int sig, \_\_sighandler\_t handler)

{

struct sigaction act;

struct sigaction oldact;

act.sa\_handler = handler;

sigemptyset(&act.sa\_mask);

act.sa\_flags = 0;

if (sigaction(sig, &act, &oldact) < 0)

return SIG\_ERR;

return oldact.sa\_handler;

}

void handler(int sig)

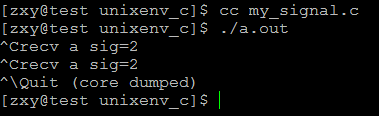
{

printf("recv a sig=%d\n", sig);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15192104-acbf6556694b490f8750c5cc2b948d00.png)

可知my\_signal与系统调用signal具有相同的效果

实例三：验证sigaction.sa\_mask效果

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig);

int main(int argc, char \*argv[])

{

struct sigaction act;

act.sa\_handler = handler;

sigemptyset(&act.sa\_mask);

sigaddset(&act.sa\_mask, SIGQUIT);

act.sa\_flags = 0;

if (sigaction(SIGINT, &act, NULL) < 0)

ERR\_EXIT("sigaction error");

struct sigaction act2;

act2.sa\_handler = handler;

sigemptyset(&act2.sa\_mask);

act2.sa\_flags = 0;

if (sigaction(SIGQUIT, &act2, NULL) < 0)

ERR\_EXIT("sigaction error");

for (;;)

pause();

return 0;

}

void handler(int sig)

{

if(sig == SIGINT){

printf("recv a SIGINT signal\n");

sleep(5);

}

if (sig == SIGQUIT)

{

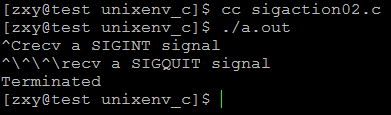
printf("recv a SIGQUIT signal\n");

}

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15192104-c4bb85c2054d41a881b929eb17efd47e.png)

可知，安装信号SIGINT时，将SIGQUIT加入到sa\_mask阻塞集中，则当SIGINT信号正在执行处理函数时，SIGQUIT信号将被阻塞，只有当SIGINT信号处理函数执行完后才解除对SIGQUIT信号的阻塞，由于SIGQUIT是不可靠信号，不支持排队，所以只递达一次

示例四：给自身发送int型数据

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

void sighandler(int signo, siginfo\_t \*info,void \*ctx);

//给自身传递信息

int main(void)

{

struct sigaction act;

act.sa\_sigaction = sighandler;

sigemptyset(&act.sa\_mask);

act.sa\_flags = SA\_SIGINFO;//信息传递开关

if(sigaction(SIGINT,&act,NULL) == -1){

perror("sigaction error");

exit(EXIT\_FAILURE);

}

sleep(2);

union sigval mysigval;

mysigval.sival\_int = 100;

if(sigqueue(getpid(),SIGINT,mysigval) == -1){

perror("sigqueue error");

exit(EXIT\_FAILURE);

}

return 0;

}

void sighandler(int signo, siginfo\_t \*info,void \*ctx)

{

//以下两种方式都能获得sigqueue发来的数据

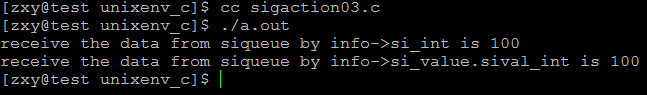
printf("receive the data from siqueue by info->si\_int is %d\n",info->si\_int);

printf("receive the data from siqueue by info->si\_value.sival\_int is %d\n",info->si\_value.sival\_int);

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15192105-d25f4f7d88e84f7987ca456e43e6da72.png)

示例五：进程间传递数据

接收端：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

void sighandler(int signo, siginfo\_t \*info,void \*ctx);

//给自身传递信息

int main(void)

{

struct sigaction act;

act.sa\_sigaction = sighandler;

sigemptyset(&act.sa\_mask);

act.sa\_flags = SA\_SIGINFO;//信息传递开关

if(sigaction(SIGINT,&act,NULL) == -1){

perror("sigaction error");

exit(EXIT\_FAILURE);

}

for(; ;){

printf("waiting a SIGINT signal....\n");

pause();

}

return 0;

}

void sighandler(int signo, siginfo\_t \*info,void \*ctx)

{

//以下两种方式都能获得sigqueue发来的数据

printf("receive the data from siqueue by info->si\_int is %d\n",info->si\_int);

printf("receive the data from siqueue by info->si\_value.sival\_int is %d\n",info->si\_value.sival\_int);

}

[复制代码](javascript:void(0);)

发送端：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

int main(int argc, char \*\*argv)

{

if(argc != 2){

fprintf(stderr,"usage:%s pid\n",argv[0]);

exit(EXIT\_FAILURE);

}

pid\_t pid = atoi(argv[1]);

sleep(2);

union sigval mysigval;

mysigval.sival\_int = 100;

printf("sending SIGINT signal to %d......\n",pid);

if(sigqueue(pid,SIGINT,mysigval) == -1){

perror("sigqueue error");

exit(EXIT\_FAILURE);

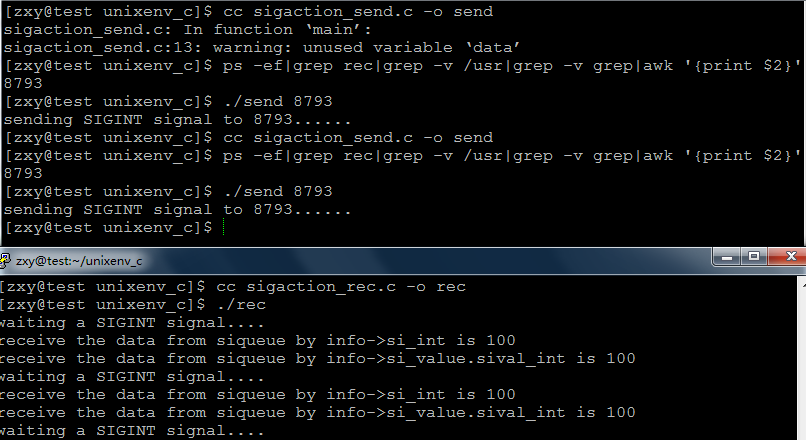
}

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15192106-58abd5564606457cabe769728c7405c2.png)

由图可知接收成功

# [linux系统编程之信号（七）：被信号中断的系统调用和库函数处理方式](http://www.cnblogs.com/mickole/p/3191832.html)

    一些IO系统调用执行时, 如 read 等待输入期间, 如果收到一个信号,系统将中断read, 转而执行信号处理函数. 当信号处理返回后, 系统遇到了一个问题: 是重新开始这个系统调用, 还是让系统调用失败?早期UNIX系统的做法是, 中断系统调用, 并让系统调用失败, 比如read返回 -1, 同时设置 errno 为 EINTR中断了的系统调用是没有完成的调用, 它的失败是临时性的, 如果再次调用则可能成功, 这并不是真正的失败, 所以要对这种情况进行处理, 典型的方式为:

[复制代码](javascript:void(0);)

while (1) {

n = read(fd, buf, BUFSIZ);

if (n == -1 && errno != EINTR) {

printf("read error\n");

break;

}

if (n == 0) {

printf("read done\n");

break;

}

}

[复制代码](javascript:void(0);)

这样做逻辑比较繁琐, 事实上, 我们可以从信号的角度来解决这个问题,  安装信号的时候, 设置 SA\_RESTART属性, 那么当信号处理函数返回后, 被该信号中断的系统调用将自动恢复.

示例程序：

[复制代码](javascript:void(0);)

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <error.h>

#include <string.h>

#include <unistd.h>

void sig\_handler(int signum)

{

printf("in handler\n");

sleep(1);

printf("handler return\n");

}

int main(int argc, char \*\*argv)

{

char buf[100];

int ret;

struct sigaction action, old\_action;

action.sa\_handler = sig\_handler;

sigemptyset(&action.sa\_mask);

action.sa\_flags = 0;

/\* 版本1:不设置SA\_RESTART属性

\* 版本2:设置SA\_RESTART属性 \*/

//action.sa\_flags |= SA\_RESTART;

sigaction(SIGINT, NULL, &old\_action);

if (old\_action.sa\_handler != SIG\_IGN) {

sigaction(SIGINT, &action, NULL);

}

bzero(buf, 100);

ret = read(0, buf, 100);

if (ret == -1) {

perror("read");

}

printf("read %d bytes:\n", ret);

printf("%s\n", buf);

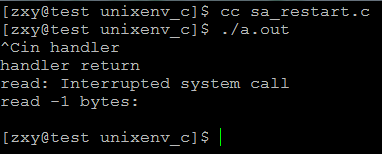
return 0;

}

[复制代码](javascript:void(0);)

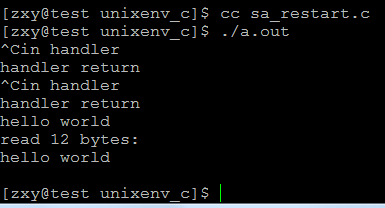
当sa\_flags不设置：SA\_RESTART时：

结果：

[](http://images.cnitblog.com/blog/529981/201307/15194226-27f5b049479a4698b9b7f69797ee5486.png)

设置后：

当被中断后，重新执行

[](http://images.cnitblog.com/blog/529981/201307/15194226-5cc76387dcf747ee85ca2534821e4602.png)

man帮助说明：

**Interruption of system calls and library functions by signal handlers**

If a signal handler is invoked while a system call or library

function call is blocked, then either:

\* the call is automatically restarted after the signal handler

returns; or

\* the call fails with the error **EINTR**.

Which of these two behaviors occurs depends on the interface and

whether or not the signal handler was established using the

**SA\_RESTART** flag (see [sigaction(2)](http://man7.org/linux/man-pages/man2/sigaction.2.html)). The details vary across UNIX

systems; below, the details for Linux.

If a blocked call to one of the following interfaces is interrupted

by a signal handler, then the call will be automatically restarted

after the signal handler returns if the **SA\_RESTART** flag was used;

otherwise the call will fail with the error **EINTR**:

\* [read(2)](http://man7.org/linux/man-pages/man2/read.2.html), [readv(2)](http://man7.org/linux/man-pages/man2/readv.2.html), [write(2)](http://man7.org/linux/man-pages/man2/write.2.html), [writev(2)](http://man7.org/linux/man-pages/man2/writev.2.html), and [ioctl(2)](http://man7.org/linux/man-pages/man2/ioctl.2.html) calls on

"slow" devices. A "slow" device is one where the I/O call may

block for an indefinite time, for example, a terminal, pipe, or

socket. (A disk is not a slow device according to this

definition.) If an I/O call on a slow device has already

transferred some data by the time it is interrupted by a signal

handler, then the call will return a success status (normally,

the number of bytes transferred).

\* [open(2)](http://man7.org/linux/man-pages/man2/open.2.html), if it can block (e.g., when opening a FIFO; see

[fifo(7)](http://man7.org/linux/man-pages/man7/fifo.7.html)).

\* [wait(2)](http://man7.org/linux/man-pages/man2/wait.2.html), [wait3(2)](http://man7.org/linux/man-pages/man2/wait3.2.html), [wait4(2)](http://man7.org/linux/man-pages/man2/wait4.2.html), [waitid(2)](http://man7.org/linux/man-pages/man2/waitid.2.html), and [waitpid(2)](http://man7.org/linux/man-pages/man2/waitpid.2.html).

\* Socket interfaces: [accept(2)](http://man7.org/linux/man-pages/man2/accept.2.html), [connect(2)](http://man7.org/linux/man-pages/man2/connect.2.html), [recv(2)](http://man7.org/linux/man-pages/man2/recv.2.html), [recvfrom(2)](http://man7.org/linux/man-pages/man2/recvfrom.2.html),

[recvmsg(2)](http://man7.org/linux/man-pages/man2/recvmsg.2.html), [send(2)](http://man7.org/linux/man-pages/man2/send.2.html), [sendto(2)](http://man7.org/linux/man-pages/man2/sendto.2.html), and [sendmsg(2)](http://man7.org/linux/man-pages/man2/sendmsg.2.html), unless a

timeout has been set on the socket (see below).

\* File locking interfaces: [flock(2)](http://man7.org/linux/man-pages/man2/flock.2.html) and [fcntl(2)](http://man7.org/linux/man-pages/man2/fcntl.2.html) **F\_SETLKW**.

\* POSIX message queue interfaces: [mq\_receive(3)](http://man7.org/linux/man-pages/man3/mq_receive.3.html),

[mq\_timedreceive(3)](http://man7.org/linux/man-pages/man3/mq_timedreceive.3.html), [mq\_send(3)](http://man7.org/linux/man-pages/man3/mq_send.3.html), and [mq\_timedsend(3)](http://man7.org/linux/man-pages/man3/mq_timedsend.3.html).

\* [futex(2)](http://man7.org/linux/man-pages/man2/futex.2.html) **FUTEX\_WAIT** (since Linux 2.6.22; beforehand, always

failed with **EINTR**).

\* POSIX semaphore interfaces: [sem\_wait(3)](http://man7.org/linux/man-pages/man3/sem_wait.3.html) and [sem\_timedwait(3)](http://man7.org/linux/man-pages/man3/sem_timedwait.3.html)

(since Linux 2.6.22; beforehand, always failed with **EINTR**).

The following interfaces are never restarted after being interrupted

by a signal handler, regardless of the use of **SA\_RESTART**; they always

fail with the error **EINTR** when interrupted by a signal handler:

\* Socket interfaces, when a timeout has been set on the socket

using [setsockopt(2)](http://man7.org/linux/man-pages/man2/setsockopt.2.html): [accept(2)](http://man7.org/linux/man-pages/man2/accept.2.html), [recv(2)](http://man7.org/linux/man-pages/man2/recv.2.html), [recvfrom(2)](http://man7.org/linux/man-pages/man2/recvfrom.2.html), and

[recvmsg(2)](http://man7.org/linux/man-pages/man2/recvmsg.2.html), if a receive timeout (**SO\_RCVTIMEO**) has been set;

[connect(2)](http://man7.org/linux/man-pages/man2/connect.2.html), [send(2)](http://man7.org/linux/man-pages/man2/send.2.html), [sendto(2)](http://man7.org/linux/man-pages/man2/sendto.2.html), and [sendmsg(2)](http://man7.org/linux/man-pages/man2/sendmsg.2.html), if a send

timeout (**SO\_SNDTIMEO**) has been set.

\* Interfaces used to wait for signals: [pause(2)](http://man7.org/linux/man-pages/man2/pause.2.html), [sigsuspend(2)](http://man7.org/linux/man-pages/man2/sigsuspend.2.html),

[sigtimedwait(2)](http://man7.org/linux/man-pages/man2/sigtimedwait.2.html), and [sigwaitinfo(2)](http://man7.org/linux/man-pages/man2/sigwaitinfo.2.html).

\* File descriptor multiplexing interfaces: [epoll\_wait(2)](http://man7.org/linux/man-pages/man2/epoll_wait.2.html),

[epoll\_pwait(2)](http://man7.org/linux/man-pages/man2/epoll_pwait.2.html), [poll(2)](http://man7.org/linux/man-pages/man2/poll.2.html), [ppoll(2)](http://man7.org/linux/man-pages/man2/ppoll.2.html), [select(2)](http://man7.org/linux/man-pages/man2/select.2.html), and [pselect(2)](http://man7.org/linux/man-pages/man2/pselect.2.html).

\* System V IPC interfaces: [msgrcv(2)](http://man7.org/linux/man-pages/man2/msgrcv.2.html), [msgsnd(2)](http://man7.org/linux/man-pages/man2/msgsnd.2.html), [semop(2)](http://man7.org/linux/man-pages/man2/semop.2.html), and

[semtimedop(2)](http://man7.org/linux/man-pages/man2/semtimedop.2.html).

\* Sleep interfaces: [clock\_nanosleep(2)](http://man7.org/linux/man-pages/man2/clock_nanosleep.2.html), [nanosleep(2)](http://man7.org/linux/man-pages/man2/nanosleep.2.html), and

[usleep(3)](http://man7.org/linux/man-pages/man3/usleep.3.html).

\* [read(2)](http://man7.org/linux/man-pages/man2/read.2.html) from an [inotify(7)](http://man7.org/linux/man-pages/man7/inotify.7.html) file descriptor.

\* [io\_getevents(2)](http://man7.org/linux/man-pages/man2/io_getevents.2.html).

The [sleep(3)](http://man7.org/linux/man-pages/man3/sleep.3.html) function is also never restarted if interrupted by a

handler, but gives a success return: the number of seconds remaining

to sleep.

# [linux系统编程之信号（八）：三种时间结构及定时器setitimer()详解](http://www.cnblogs.com/mickole/p/3191977.html)

### 一，三种时间结构

time\_t://seconds

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

struct timespec {

time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

### 二，setitimer()

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：

#include <sys/time.h>

int getitimer(int which, struct itimerval \*curr\_value);   
int setitimer(int which, const struct itimerval \*new\_value,struct itimerval \*old\_value);

参数:

* 第一个参数which指定定时器类型
* 第二个参数是结构itimerval的一个实例，结构itimerval形式
* 第三个参数可不做处理。

返回值:成功返回0失败返回-1

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：

TIMER\_REAL：按实际时间计时，计时到达将给进程发送SIGALRM信号。

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。

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; /\* 第一次之后每隔多长时间 \*/

        struct timeval it\_value; /\* 第一次调用要多长时间 \*/

};

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。

示例一：

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#include <sys/time.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig)

{

printf("recv a sig=%d\n", sig);

}

int main(int argc, char \*argv[])

{

if (signal(SIGALRM, handler) == SIG\_ERR)

ERR\_EXIT("signal error");

struct timeval tv\_interval = {1, 0};

struct timeval tv\_value = {5, 0};

struct itimerval it;

it.it\_interval = tv\_interval;

it.it\_value = tv\_value;

setitimer(ITIMER\_REAL, &it, NULL);

for (;;)

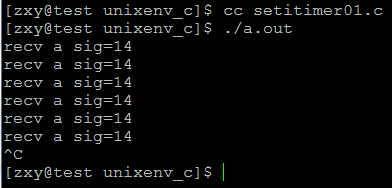
pause();

return 0;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15205715-850824a3b5ba4b40b6a485269edee00b.png)

可以看到第一次发送信号是在5s以后，之后每隔一秒发送一次信号

示例二：获得产生时钟信号的剩余时间

[复制代码](javascript:void(0);)

#include <unistd.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <signal.h>

#include <sys/time.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

int main(int argc, char \*argv[])

{

struct timeval tv\_interval = {1, 0};

struct timeval tv\_value = {1, 0};

struct itimerval it;

it.it\_interval = tv\_interval;

it.it\_value = tv\_value;

setitimer(ITIMER\_REAL, &it, NULL);

int i;

for (i=0; i<10000; i++);

//第一种方式获得剩余时间

struct itimerval oit;

setitimer(ITIMER\_REAL, &it, &oit);//利用oit获得剩余时间产生时钟信号

printf("%d %d %d %d\n", (int)oit.it\_interval.tv\_sec, (int)oit.it\_interval.tv\_usec, (int)oit.it\_value.tv\_sec, (int)oit.it\_value.tv\_usec);

//第二种方式获得剩余时间

//getitimer(ITIMER\_REAL, &it);

//printf("%d %d %d %d\n", (int)it.it\_interval.tv\_sec, (int)it.it\_interval.tv\_usec, (int)it.it\_value.tv\_sec, (int)it.it\_value.tv\_usec);

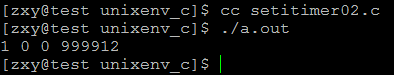
return 0;

}

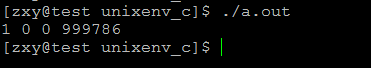
[复制代码](javascript:void(0);)

结果：

用第一种方式：

[](http://images.cnitblog.com/blog/529981/201307/15205717-a9ede669cca74f86bd4f82740b910632.png)

用第二种方式：利用getitimer在不重新设置时钟的情况下获取剩余时间

[](http://images.cnitblog.com/blog/529981/201307/15205717-8aec34f6cc334c789f9d4a450edee5a9.png)

剩余时间是指：距离下一次调用定时器产生信号所需时间，这里由于for循环不到一秒就执行完，定时器还来不及产生时钟信号，所以有剩余时间

示例三：每隔一秒发出一个SIGALRM，每隔0.5秒发出一个SIGVTALRM信号

[复制代码](javascript:void(0);)

#include <signal.h>

#include <unistd.h>

#include <stdio.h>

#include <sys/time.h>

void sigroutine(int signo)

{

switch (signo) {

case SIGALRM:

printf("Catch a signal -- SIGALRM\n ");

break;

case SIGVTALRM:

printf("Catch a signal -- SIGVTALRM\n ");

break;

}

return;

}

int main()

{

struct itimerval value,value2;

printf("process id is %d\n ",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;

setitimer(ITIMER\_REAL, &value,NULL);

value2.it\_value.tv\_sec = 0;

value2.it\_value.tv\_usec = 500000;

value2.it\_interval.tv\_sec = 0;

value2.it\_interval.tv\_usec = 500000;

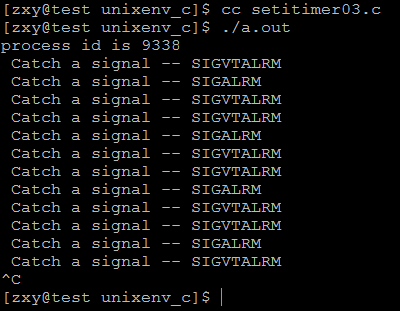
setitimer(ITIMER\_VIRTUAL, &value2,NULL);

for (;;) ;

}

[复制代码](javascript:void(0);)

结果：

[](http://images.cnitblog.com/blog/529981/201307/15205718-5dde46ba2d0e45fe9b1d5b5f20cc34f0.png)

可知确实是没两次SIGVTALRM一次SIGALRM