# 第6章 理解信号机制

从生活的角度来理解，信号的本质是异步。不得不承认，这个世界是异步的。试想如下的场景，我正在公司里面聚精会神地工作，场景切换到办公楼的入口，快递小哥儿掏出手机，拨打了我的手机（发送信号），移动通信很给力，很快我的手机铃声就响了（递送信号），我停下手头的工作，从口袋里掏出手机，开始接听电话（执行信号处理函数）。这个世界就是这样，我无法预测快递何时给我打电话让我签收快递。我刚收完快递，准备继续工作，突然，测试人员找我定位一个bug......这就是世界的每个角落不断发生的事情，异步，事情可能会发生，你无法预测它们何时发生，而它们发生时，我需要作出响应。更深入一点，如果我参加非常重要的会议，我可能会屏蔽快递小哥儿的电话（如果我知道他的电话的话），这已经是信号的高级应用（sigprocmask）了。

从计算机的角度理解，信号是一种软件中断，用来处理异步事件。内核递送这些异步事件到某个进程，告诉进程知道某个特殊事件发生了。这些异步事件，可能来源硬件，比如访问了非法的内存地址，除以0；可能来源于用户的输入，比如我在键盘上敲击了Ctrl+c；有些可能来源于另一个进程，有些甚至来源于进程自身。

信号的本质是一种进程间的通信，一个进程可以向另一个进程发送信号，内核至少传递了signal number这个字段。实际上，通信的内容可以不止是signal number。

Signal是Unix家族里一个古老的通信机制，由于传统的信号机制有一些的弊端，很多Unix/Linux编程人员并不喜欢信号机制。然而，随着时间的推移，由于可靠信号的引入，信号的可靠性得到了改善，同时功能方面也有增强，除了信号编号值可以递送个目标进程，还可以发送一些附加信息。可以说，一个服务器端的程序，没有信号处理部分，是不完整的。接下来，从历史出发，梳理signal的前世和今生，学习Linux对信号的支持。

## 6.1信号的完整生命周期

前文提到过，信号的本质是一种进程间的通信。进程之间约定好：如果发生了某件事情T（trigger），那么向目标进程（destination process）发送某特定信号X，而目标进程看到X，就意识到，T事件发生了，目标进程就执行了相应的动作A(action)。

接下来以配置文件改变为例，描述这个过程。很多大型的应用都有配置文件，如果配置文件发生改变，需要通知进程重新加载配置。一般而言，很多程序会默认采用SIGHUP信号来通知目标进程重新加载配置文件。

目标进程首先约定，只要收到SIGHUP，就执行重新加载配置文件的动作。这个行为，叫做信号的安装（installation），或者信号处理函数的注册。安装好了之后，因为信号是异步事件，不知道何时会发生，所以目标进程干自己的事情。某年某月的某天，管理员突然改变了配置文件，管理员想通知这个目标进程，所以管理员就向目标进程发送了信号。他可能在终端执行了kill -SIGHUP命令 ,也可能他调用了C的API，whatever，信号产生(generation)了。这时候，Linux内核收到了产生的信号，Linux内核就会在目标进程的进程描述符里记录一笔：收到信号SIGHUP一枚。Linux内核负责进程的调度，在合适的时机，讲信号递送（deliver）给进程，使进程执行安装时指定的信号处理函数（signal handler）。在内核收到信号，但是还没有递送给目标进程的这一段时间，信号处于挂起状态，被称为挂起（pending）信号,也称为未决信号。

一个典型的信号是如上所述处理的，但是实际情况要复杂的多，还有很多不同的情况需要考虑：

* 目标进程正在执行关键代码，不能被信号中断，需要阻塞某信号，那么在这期间，信号就不允许递送到进程，直到目标进程解除阻塞；
* 内核发现同一个信号已经存在，内核如何处理这种重复的信号，排队还是丢弃；
* 内核递送信号的时候，发现已有多个不同的信号挂起，内核应该优先递送哪个信号；
* 多线程的进程，如果向该进程发送信号，应该由哪个线程负责响应。

这些问题，在接下来的章节中会逐渐解决。

进程收到信号之后，会执行那些操作呢？这取决于信号处理函数：

* 显式地忽略信号：即内核将会丢弃该信号，信号不会对进程产生任何影响，进程也感知不到信号的出现
* 终止进程：很多信号的默认处理是终止进程，即将进程杀死。
* 生成Core Dump文件并终止进程：进程被杀死，并且产生核心转储文件。核心转储文件记录了进程死亡现场的信息。用户可以使用核心转储文件来调试，分析进程死亡的原因。
* 停止进程：停止进程不同于终止进程，终止进程，进程已经死亡，但是停止进程仅仅是使进程暂停，将进程状态设置成TASK\_STOPPED，一旦收到恢复执行的信号，进程还可以继续执行。SIGSTOP信号，SIGTSTP信号都可以使进程停止。
* 恢复进程的执行：和停止进程对应，某些信号可以使进程恢复执行。SIGCONT信号有此功效。
* 捕捉信号，执行定制的信号处理函数：信号一般都有默认的行为，但是用户可以通过注册信号处理函数，改变信号的默认响应，执行用户定制的响应。需要强调的是，并不是所有信号都可以捕捉，改变信号的默认响应，SIGKILL和SIGSTOP不可捕捉，注册信号处理函数。

6.2信号的分类

在Linux世界中，信号有哪些？

在Linux的shell终端，执行kill -l，我们可以看到所有的信号：

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR

31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3

38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8

43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13

48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12

53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7

58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2

63) SIGRTMAX-1 64) SIGRTMAX

这些信号，可以分成两类：

* 可靠信号
* 不可靠信号

[1,31]之间的所有信号，都被称为不可靠信号。

这些信号都是从传统的Unix继承而来。早期的Unix信号的机制并不完备，在实践过程中暴露了很多弊端。因此把这些出现比较早的信号值在[1,31]之间的信号称之为不可靠信号，所谓不可靠信号，是指信号可能会被丢失。

[SIGRTMIN,SIGRTMAX]之间的信号，我们称为可靠信号。

这些信号是随着时间的流逝，人们意识到原有的信号机制存在弊端而引入的。但是[1,31]之间的这些存在已久，在很多应用中广泛使用，所以只能新增信号，这些信号就是我们今天看到的[SIGRTMIN,SIGRTMAX]这些信号，它们被称为可靠信号。

对于信号有初步了解，知道signal和sigaction函数接口的同学可能会产生误解，认为用signal安装的信号，kill（或者tkill）发送的信号，就是不可靠信号，用sigaction安装的信号，用sigqueue的发送的信号，就是可靠信号。这种理解是错误的。信号的可靠与否，完全取决于信号的值。与采用那种安装方式和信号发送方式无关。

可靠信号和不可靠信号，之所以存在差异，根本原因是收到信号的后，内核的反应不同：

对于不可靠信号，内核内部维护有位图来记录是否有该信号处于挂起状态，如果收到某不可靠信号，内核发现已经存在该信号处于未决状态，内核会简单的丢弃该信号。因此不可靠信号，信号可能会丢失，即内核递送给目标进程的次数，可能小于信号发生的次数。

对于可靠信号，内核内部维护有队列，如果收到可靠信号，内核会将信号挂到相应队列，因此不会丢失。严格说来，内核也设有上限，挂起信号的个数也不能无限制的增大。

从kill –l列出的信号中，细心的同学可以看出，少了32号信号和33号信号。这两个信号（SIGCANCEL和SIGSETXID）被NPTL这个线程库征用了，用来实现线程的取消。从内核层来说，32号信号应该是最小的实时信号（SIGRTMIN），但是由于32和33被glibc内部征用了，所以glibc将SIGRTMIN设置成了34号信号。

对于传统的信号，POSIX标准规定了每种信号的默认行为，可以通过man 7 signal查看手册，获取到每种信号的默认行为，此处就不在赘述。

## 6.3传统信号的特点

前文就提到，signal是一个古老的机制，早期的信号在使用过程中，暴露出一些弊端，那么早期的信号机制有什么弊端，表现出什么样的行为模式？今天Linux下的glibc提供的信号函数是否解决了这些弊端，它又表现出什么样的行为模式？

传统的signal机制，分为System V风格的signal和BSD风格的signal。

除了标准的linux signal的API：

#include <signal.h>

typedef void (\*sighandler\_t)(int);

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

Linux还提供了如下两个接口，供我们考古，探查signal机制的演化：

#include <signal.h>

typedef void (\*sighandler\_t)(int);

sighandler\_t sysv\_signal(int signum, sighandler\_t handler);

sighandler\_t bsd\_signal(int signum, sighandler\_t handler)

接下来用实验的方法，测试各种不同信号机制表现出来的行为模式，来体会传统信号的特点以及弊端，以及学习Linux下glibc提供的signal函数的行为特性。

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <string.h>

#include <errno.h>

#include <sys/syscall.h>

#define MSG "OMG , I catch the signal SIGINT\n"

#define MSG\_END "OK,finished process signal SIGINT\n"

int do\_heavy\_work()

{

int i ;

int k;

srand(time(NULL));

for(i = 0 ; i < 100000000;i++)

{

k = rand()%1234589;

}

return 0;

}

void signal\_handler(int signo)

{

write(2,MSG,strlen(MSG));

do\_heavy\_work();

write(2,MSG\_END,strlen(MSG\_END));

}

int main()

{

char input[1024] = {0};

#if defined SYSTEMCALL\_SIGNAL\_API

*if(syscall(SYS\_signal ,SIGINT,signal\_handler) == -1)*

#elif defined SYSTEMV\_SIGNAL\_API

*if(sysv\_signal(SIGINT,signal\_handler) == -1)*

#elif defined BSD\_SIGNAL\_API

*if(sysv\_signal(SIGINT,signal\_handler) == -1)*

#else

*if(signal(SIGINT,signal\_handler) == SIG\_ERR)*

#endif

{

fprintf(stderr,"signal failed\n");

return -1;

}

printf("input a string:\n");

if(fgets(input,sizeof(input),stdin)== NULL)

{

fprintf(stderr,"fgets failed(%s)\n",strerror(errno));

return -2;

}

else

{

printf("you entered:%s",input);

}

return 0;

}

这个测试程序的核心是斜体的地方，这个函数分别采用了Linux操作系统提供的signal系统调用，System V风格的sysv\_signal，BSD风格的bsd\_signal，还有glibc提供的标准API signal函数。我们来分别体会他们之间的不同。

gcc -o systemcall\_signal -DSYSTEMCALL\_SIGNAL\_API signal\_comp.c

gcc -o sysv\_signal -DSYSTEMV\_SIGNAL\_API signal\_comp.c

gcc -o bsd\_signal -DBSD\_SIGNAL\_API signal\_comp.c

gcc -o glibc\_signal signal\_comp.c

我们分别生成了四种风格的测试程序。我们可以开始实验他们的特性了。

### 6.3.1 信号的ONESHOT特性

传统的System V风格的signal，信号处理函数是一次性的，信号deliver之后，信号处理函数会变成默认值SIG\_DFL。

Linux的系统调用signal（注意是系统调用，而不是glibc的signal函数）也有这个弊端。

manu@manu-hacks:~/code/c/self/signal$ ./sysv\_signal

input a string:

hello

you entered:hello

manu@manu-hacks:~/code/c/self/signal$ ./sysv\_signal

input a string:

hello^COMG , I catch the signal SIGINT

^C

manu@manu-hacks:~/code/c/self/signal$

可以看到第一次实验的时候，输入一个字符串，敲击回车，正常显示了输入的字符串。第二次输入结束之前，敲击了Ctrl+c，系统向进程发送SIGINT信号。执行了信号处理函数（打印出了OMG，I catch the signal SIGINT），但是故意将信号处理函数设计的非常的耗时，在执行信号处理函数期间，我再次发送SIGINT信号，进程退出了。

System V风格的信号处理机制，是一次性的，内核信号deliver出去后，信号处理函数恢复成默认信号SIG\_DFL。因为SIGINT的默认处理是终止进程，所以进程退出了。

这种风格，Kernel中有个很形象的宏，描述这种行为模式：SA\_ONESHOT



就像这种老式的单发手枪，每次射击之后，都要重新上子弹。System V风格的singal处理机制就是如此，信号处理函数触发之后，要想重复触发，必须再次安装信号处理函数。

linux系统调用也是如此，信号处理函数是一次性的：

manu@manu-hacks:~/code/c/self/signal$ ./systemcall\_signal

input a string:

hello

you entered:hello

manu@manu-hacks:~/code/c/self/signal$ ./systemcall\_signal

input a string:

hello^COMG , I catch the signal SIGINT

^C

manu@manu-hacks:~/code/c/self/signal$

内核如何做到的？

我以signal系统调用为例，介绍下ONESHOT是如何做到的？

SYSCALL\_DEFINE2(signal, int, sig, \_\_sighandler\_t, handler)

{

struct k\_sigaction new\_sa, old\_sa;

int ret;

new\_sa.sa.sa\_handler = handler;

new\_sa.sa.sa\_flags = SA\_ONESHOT | SA\_NOMASK;

sigemptyset(&new\_sa.sa.sa\_mask);

ret = do\_sigaction(sig, &new\_sa, &old\_sa);

return ret ? ret : (unsigned long)old\_sa.sa.sa\_handler;

}

我们可以看到，系统调用signal，本质调用了do\_sigaction。sysv\_signal也好，bsd\_signal也好，glibc的signal函数也好，包括后起之秀sigaction函数，最终是都会调用do\_sigaction。内核如何区分中这些函数，让他们各司其职，做自己应该做的事情呢？就是通过sa\_flags标志位，通过一个个的标志位来控制一个个的行为。

对于signal系统调用而言，有两个标志位，SA\_ONESHOT和SA\_NOMASK，我们本节讲述的是SA\_ONESHOT。

前文讲解signal的生命周期的时候，提到一个进程可能有多个信号处于挂起状态，内核会在合适的时机，选择一个信号deliver给进程，令进程执行信号处理函数。我们所谓的SA\_ONESHOT，将信号处理函数恢复成默认值就是选择信号deliver给进程时完成的。

int get\_signal\_to\_deliver(siginfo\_t \*info,

struct k\_sigaction \*return\_ka,

struct pt\_regs \*regs,

void \*cookie)

{

...

if (ka->sa.sa\_handler == SIG\_IGN) /\* Do nothing. \*/

continue;

if (ka->sa.sa\_handler != SIG\_DFL) {

/\* Run the handler. \*/

\*return\_ka = \*ka;

if (ka->sa.sa\_flags & SA\_ONESHOT)

ka->sa.sa\_handler = SIG\_DFL;

break; /\* will return non-zero "signr" value \*/

}

...

}

sysv\_signal函数是如何加上SA\_ONESHOT标志位的呢？我们看下sysv\_signal如何走到内核。

我们strace执行sysv\_signal，我们可以看到调用如下系统调用：

rt\_sigaction(SIGINT, {0x8048756, [], SA\_INTERRUPT|SA\_NODEFER|SA\_RESETHAND}, {SIG\_DFL, [], 0}, 8) = 0

我们看到了，有SA\_RESETHAND标志位，这个标志位和SA\_ONESHOT是一个含义。

#define SA\_ONESHOT SA\_RESETHAND

BSD风格的signal和glibc的signal函数已经不存在这个ONESHOT的问题。

manu@manu-hacks:~/code/c/self/signal$ ./bsd\_signal

input a string:

hello^COMG , I catch the signal SIGINT

^COK,finished process signal SIGINT

OMG , I catch the signal SIGINT

OK,finished process signal SIGINT

^COMG , I catch the signal SIGINT

OK,finished process signal SIGINT

manu@manu-hacks:~/code/c/self/signal$ ./glibc\_signal

input a string:

hello^COMG , I catch the signal SIGINT

^COK,finished process signal SIGINT

OMG , I catch the signal SIGINT

^COK,finished process signal SIGINT

OMG , I catch the signal SIGINT

OK,finished process signal SIGINT

通过strace bsd\_signal和glibc\_signal 我们可以看出来，他们两者都没有SA\_ONESHOT的标志位。

rt\_sigaction(SIGINT, {0x8048736, [INT], SA\_RESTART}, {SIG\_DFL, [], 0}, 8) = 0

在执行sysv\_signal期间因为我们的信号处理函数耗时非常长，在执行信号处理函数的期间，进程收到了另外一个信号，并且执行了信号处理函数SIG\_DFL，即进程退出。这反映出一个问题，对于System V风格的signal机制，执行信号处理函数期间，并没有屏蔽相应的信号。

### 6.3.2 信号执行时的屏蔽自身特性

在信号处理函数期间，很有可能收到其他信号，当然也有可能再次收到自身这个信号。如果在处理A信号期间，再次收到A信号，会如何处理呢？

传统的System V信号，信号处理期间，不会屏蔽对应的信号。这会引起信号处理函数的重入。这个算是传统的System V信号的另一个弊端。BSD信号处理机制修正的这个弊端。当然了，BSD信号处理机制只是屏蔽了当前信号，并没有定制化的屏蔽某些信号。

我们来比较下System V和BSD 的signal机制的区别：

rt\_sigaction(SIGINT, {0x8048756, [], SA\_INTERRUPT|SA\_NODEFER|SA\_RESETHAND}, {SIG\_DFL, [], 0}, 8) = 0

rt\_sigaction(SIGINT, {0x8048736, [INT], SA\_RESTART}, {SIG\_DFL, [], 0}, 8) = 0

可以看到，rt\_sigaction有4个入参，第一个是信号值，表示你为哪个信号注册信号处理函数，第二个参数和第三个参数的类型是一致的，是struct sigaction类型：

SYSCALL\_DEFINE4(rt\_sigaction, int, sig,

const struct sigaction \_\_user \*, act,

struct sigaction \_\_user \*, oact,

size\_t, sigsetsize)

sigaction结构体，是个体系结构相关的变量，同时内核中的定义和glibc中的定义略有不同，但是一般说来，都会有如下的结构：

struct sigaction {

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

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

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

上述数据结构中，有一个sigset\_t类型的变量sa\_mask。sigset\_t类型本质是个位图，我们有64种不同的信号，所以我们需要64个bit来表示每个信号。在rt\_sigaction系统调用中，sigset\_t位图表示的是，执行信号处理函数期间，应该暂时屏蔽那些信号。所谓屏蔽，就是纵然收到这些信号，将这些信号挂起，待信号处理函数执行完毕之后，再来处理这些信号。

从上面的描述可以看出，BSD风格的信号处理机制，在安装信号的时候，会将自身这个信号添加到信号处理函数的屏蔽集合中，不会出现处理A信号的时候，再次收到A信号，而去中断，先执行后收到的A信号。简单的说，就是不会嵌套了。而System V 风格的信号处理期间没有屏蔽任何信号，换句话说，执行信号处理函数期间，可以被任意信号中断。从前面的实验可以看出，BSD style的信号处理函数，OMG , I catch the signal SIGINT，以及OK,finished process signal SIGINT是成对出现的，不可能出现连续两个OMG , I catch the signal SIGINT，原因就是SIGINT信号在信号处理函数执行期间，被暂时屏蔽了。

内核是如何做到这一点的？

完整的signal deliver的流程大致如此：

内核首先调用get\_signal\_to\_deliver在挂起信号的集合中选择一个信号，deliver给进程，选择完毕后，调用handler\_signal函数：

void signal\_delivered(int sig, siginfo\_t \*info, struct k\_sigaction \*ka,

struct pt\_regs \*regs, int stepping)

{

sigset\_t blocked;

……

clear\_restore\_sigmask();

*sigorsets(&blocked, &current->blocked, &ka->sa.sa\_mask);*

*if (!(ka->sa.sa\_flags & SA\_NODEFER))*

*sigaddset(&blocked, sig);*

set\_current\_blocked(&blocked);

tracehook\_signal\_handler(sig, info, ka, regs, stepping);

}

比较核心的两行代码是讲当前信号处理函数的屏蔽信号集加入到blocked中，同时将当前信号也加入到屏蔽信号集合中，除非显式的指定SA\_NODEFER，才不屏蔽自身信号值。

System V style的signal机制为何会出现不屏蔽自身信号？原因就是sysv\_signal函数，调用rt\_sigaction系统调用时加上了SA\_NODEFER标志位：

rt\_sigaction(SIGINT, {0x8048756, [], SA\_INTERRUPT|**SA\_NODEFER**|SA\_RESETHAND}, {SIG\_DFL, [], 0}, 8) = 0

通过实验我们可以看到，BSD style的信号处理机制在信号处理函数执行期间，会将自身信号加入到屏蔽集合，glibc 的signal也会如此，目前只有System V style的信号处理机制存在此弊端。

### 6.3.3 信号中断系统调用的重启特性

很多系统调用在执行期间，很可能会收到信号，进程不得不从系统调用中返回，去执行信号处理函数。对于执行时间比较久的系统调用（如wait，read等）被信号中断的可能性更会大大增加。系统调用被中断，一般返回失败，并置错误码为EINTR。如果程序员希望处理完信号之后，被中断的系统调用能够重启，不得不自行判断errno，如果发现错误码是EINTR，就重新调用系统调用。看下面的例子。

manu@manu-hacks:~/code/c/self/signal$ ./sysv\_signal

input a string:

^COMG , I catch the signal SIGINT

OK,finished process signal SIGINT

fgets failed(Interrupted system call)

可以通过strace看到，fgets调用了read系统调用，而read系统调用阻塞在等用户输入，在阻塞过程中，收到了一枚SIGINT信号，导致read系统调用被中断，返回了错误码EINTR。

Linux世界中很多系统调用都会遭遇这种情景，尤其是read，wait这种可能比较耗时的系统调用。<UNIX系统编程>>一书中很多类似的例子：

pid\_t r\_wait(int \*stat\_loc)

{

int retval;

while(((retval = wait(stat\_loc)) ==-1 && (errno == EINTR))

{

;

}

return retval;

}

这种封装，就是用来应对系统调用被信号中断的场景。当系统调用被信号中断，程序并不认为这是一种无法处理的错误，相反，程序完全可以通过重新调用系统调用，来完成程序想做的事情。

这一次，System V信号机制再次表现不佳，它会将系统调用中断，并置errno为EINTR，但是不会主动重启被信号中断的系统调用。细细想来，如果所有的系统调用都要判断返回值是否是EINTR，如果是，重启系统调用，那么程序员太累了。BSD style的signal机制完美的解决了这个问题，如果系统调用被信号中断，内核会在信号处理函数结束之后，自动重启系统调用，无需程序员显式地再次调用系统调用。

Linux作为操作系统的后期之秀，为了解决这个问题，提供了一个标志位来告诉内核，被信号中断后是否重启系统调用：SA\_RESTART。

我们看下BSD风格的signal的标志位：

rt\_sigaction(SIGINT, {0x8048736, [INT], SA\_RESTART}, {SIG\_DFL, [], 0}, 8) = 0

由于BSD风格的signal存在这个标志SA\_RESTART，fgets不会像System V的signal一样，返回错误码：

manu@manu-hacks:~/code/c/self/signal$ ./bsd\_signal

input a string:

hello^COMG , I catch the signal SIGINT

OK,finished process signal SIGINT

^COMG , I catch the signal SIGINT

OK,finished process signal SIGINT

glibc的signal函数表现如何？从我的测试程序可以看出glibc的signal函数也带上了SA\_RESTART标志位。

非常不幸的是，并不是所有的系统调用对信号中断都表现出同样的行为。正相反，某些系统调用哪怕是设置了SA\_RESTART的标志位，也绝不会自动重启。

那问题就来了，Linux下，如果信号处理函数设置了SA\_RESTART，那些阻塞型的系统调用遭到信号中断时，可以自动重启，那些系统调用是死活也无法自动重启的？

首先是设置SA\_RESTART标志位后，可以自动重启的阻塞型系统调用：

* 访问慢设备的I/O系统调用，如read()，write()，readv()，writev()，ioctl()等。对于read/write函数有以下的特点：
  + 如果被信号中断前，已经传递了部分数据，那么系统调用被中断，但是返回成功的状态，即返回成功读写的字节数
  + 如果没有任何数据完成读写，那么返回-1，并且设置errno为EINTR。
* 系统调用open()，在可能阻塞的情况下，如打开FIFO时
* 等待子进程的系统调用：wait()、waitpid()、waited()等
* 套接字相关的系统调用：accept()、accept4()、connect()、send()、sendmsg()、sendto()、recv()、recvfrom()、recvmsg()等。如果通过setsockopt函数设置了超时，那么这些系统调用也不会自动重启。
* POSIX消息队列上的进行IO操作的系统调用：mq\_receive()、mq\_send()、mq\_timedreceive()、mq\_timedsend()。
* 用于设置文件锁的系统调用：flock()、fcntl()、lockf()。
* POSIX信号量的sem\_wait()和sem\_timedwait()
* 同步线程的函数：pthread\_mutex\_lock()、pthread\_mutex\_trylock()、pthread\_mutex\_timedlock()、pthread\_cond\_wait()、pthread\_cond\_timedwait()等

其次是无论如何也不会重启的系统调用：

* 多路复用函数：poll()、epoll()、select()、pselect()
* Linux特有的epoll\_wait()、epoll\_pwait()
* Linux特有的io\_getevents()
* System V消息队列和信号量
* 睡眠型接口：sleep()、nanosleep()、clock\_nanosleep()
* 设计用来等待信号到达的接口：pause()、sigsuspend()、sigwaitinfo()、sigtimedwait()。

太多了，记不住怎么办？幸好有手册，拿不准时，可以查看手册：

man 7 signal

通过前面三小节的测试，我们可以得到下面结论：

|  |  |  |  |
| --- | --- | --- | --- |
| **信号机制** | **ONESHOT特性** | **执行期间屏蔽自身信号** | **重启系统调用** |
| **System V** | YES | NO | NO |
| **BSD** | NO | YES | YES |
| **system call** | YES | NO | NO |
| **glibc signal** | NO | YES | YES |

手册明确表示bsd\_signal没有ONESHOT特性，信号处理函数不会reset成默认值，无需重复安装信号处理函数；信号处理函数期间，自身信号会被屏蔽；系统调用被中断，会重启系统调用。这三个特性都是可以保证的，但是glibc下signal函数就不一定了，要取决于操作系统。通过man 2 signal，也可以看到signal函数的移植性并不好，这也是大多数的书籍建议不要使用signal函数的原因，建议使用更明确的sigaction来安装信号的原因。

在讨论信号的特点，可以发现，占据后发优势的Linux系统通过一些标志位和屏蔽信号集来完成某些feature的控制。

* SA\_ONESHOT （或SA\_RESERTHAND）来将信号处理函数恢复成默认值
* SA\_NODEFER（或SA\_NOMASK）会显式地告诉内核，不要讲本信号值添加进block set
* SA\_RESTART会将中断的系统调用重启，而不是返回错误码EINTR。

## 6.4 传统信号的不可靠性

在评价一个人的时候，说这个人有很多的缺点，这都是正常的，毕竟人无完人，但是说一个不可靠，性质就严重很多了。前面提到了一些传统信号的弊端，或可补救，或可忍耐，但是传统的signal机制有一个致命的缺陷：不可靠。前文也提到了，信号值落在[1,31]之间的信号，称为不可靠信号，无论采用什么样的信号安装方式（singal函数或者sigaction函数），无论我们采用什么样的信号发送方式（kill，tkill，raise，sigqueue），只要选择的信号值落在了[1,31]之内，我们就不能确定，安装的信号处理函数一定会被执行。

为了对传统的不可靠信号有直观的认识，我们看一个例子。如果我们疯狂向某个进程发送信号，我们可以查看，我们发射信号的次数，和信号被递送给进程，信号处理函数被调用的次数是否相等。信号作为一种进程间通信的方式，期望的结果是，发射信号N次，那么目标进程就要执行信号处理函数N次。对于传统信号而言，实际情况如何呢？

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <string.h>

#include <errno.h>

static int sig\_cnt[NSIG];

static volatile sig\_atomic\_t get\_SIGINT = 0;

void handler(int signo)

{

if(signo == SIGINT)

get\_SIGINT = 1;

else

sig\_cnt[signo]++;

}

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

{

int i = 0;

sigset\_t blockall\_mask ;

sigset\_t pending\_mask ;

sigset\_t empty\_mask ;

printf("%s:PID is %d\n",argv[0],getpid());

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

{

if(i == SIGKILL || i == SIGSTOP)//不可屏蔽之信号

continue;

if(signal(i,&handler) == SIG\_ERR)

{

fprintf(stderr,"signal for signo(%d) failed (%s)\n",

i,strerror(errno));

}

}

if(argc > 1)

{

int sleep\_time = atoi(argv[1]);

sigfillset(&blockall\_mask);

if(sigprocmask(SIG\_SETMASK,&blockall\_mask,NULL) == -1)

{

fprintf(stderr,"setprocmask to block all signal failed(%s)\n",

strerror(errno));

return -2;

}

printf("I will sleep %d second\n",sleep\_time);

sleep(sleep\_time);

if(sigpending(&pending\_mask) == -1)

{

fprintf(stderr,"sigpending failed(%s)\n",

strerror(errno));

return -2;

}

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

{

if(sigismember(&pending\_mask,i))

printf("signo(%d) :%s\n",i,strsignal(i));

}

sigemptyset(&empty\_mask);

if(sigprocmask(SIG\_SETMASK,&empty\_mask,NULL) == -1)

{

fprintf(stderr,"setprocmask to release all signal failed(%s)\n", strerror(errno));

return -3;

}

}

while(!get\_SIGINT)

continue ;

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

{

if(sig\_cnt[i] != 0 )

{

printf("%s:signal %d caught %d time%s\n",

argv[0],i,sig\_cnt[i],(sig\_cnt[i] >1)?"s":"");

}

}

return 0;

}

简单讲述下这个signal\_receiver进程：

如果执行时不带参数，那么进程就会原地不动，直到收到SIGINT信号为止。如果进程收到了SIGKILL，SIGSTOP之外其他信号，我们会统计收到信号的次数加1，这样，进程结束前，我们打印出来每种信号收到的次数。

如果我们有一个参数，这个参数是信号阻塞时间，在这个期间，我们讲能够阻塞的信号全部阻塞，这信号阻塞期间，虽然会有进程向signal\_receiver发送信号，但是kernel并不会立即将signal 递送给进程，在解除阻塞之后，kernel开始向signal\_receiver递送信号。

我们再准备一个发送信号的程序：

#include <stdio.h>

#include <stdlib.h>

#include <getopt.h>

#include <signal.h>

#include <string.h>

#include <errno.h>

void usage()

{

fprintf(stderr,"USAGE:\n");

fprintf(stderr,"--------------------------------\n");

fprintf(stderr,"signal\_sender pid signo times\n");

}

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

{

pid\_t pid = -1 ;

int signo = -1;

int times = -1;

int i ;

if(argc < 4 )

{

usage();

return -1;

}

pid = atol(argv[1]);

signo = atoi(argv[2]);

times = atoi(argv[3]);

if(pid <= 0 || times < 0 || signo <1 ||

signo >=64 ||signo == 32 || signo ==33)

{

usage();

return -1;

}

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

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

{

if(kill(pid,signo) == -1)

{

fprintf(stderr, "send signo(%d) to pid(%d) failed,reason(%s)\n",

signo,pid,strerror(errno));

return -2;

}

}

fprintf(stdout,"done\n");

return 0;

}

程序比较简单，接受三个参数，分别是目标进程号，信号值，和发送次数。有了这个工具，我们可以连续向目标进程signal\_receiver连续发送任意次数的信号X。

首先，signal\_receiver不阻塞信号一段时间，我们向signal\_receiver连续发送信号，看下目标进程signal\_receiver一共收到多少次：

manu@manu-hacks:~/code/c/self/signal$ ./signal\_receiver

./signal\_receiver:PID is 9937

signal for signo(32) failed (Invalid argument)

signal for signo(33) failed (Invalid argument)

我们向9937进程发送信号SIGUSR2 10000次，然后发送SIGINT信号 1次，看下signal\_receiver进程一共收到信号多少次:

manu@manu-hacks:~/code/c/self/signal$ ./signal\_sender 9937 12 10000

pid = 9937,signo = 12,times = 10000

done

manu@manu-hacks:~/code/c/self/signal$ ./signal\_sender 9937 2 1

pid = 9937,signo = 2,times = 1

done

我们可以看到signal\_receiver进程打印出：

./signal\_receiver:signal 12 caught 2499 times

我们可以看到我们发送12号信号10000次，可是signal\_receiver只收到2499次，这个2499也不是固定的，我们多执行几次，每次收到的信号次数均不相同：

./signal\_receiver:signal 12 caught 2639 times

./signal\_receiver:signal 12 caught 2511 times

我们可以看到收到的信号次数的是不一定的，但是都不等于发送信号的次数。我们再进一步，让信号接收进程，屏蔽信号一段时间，在这段时间内，我们发送信号，来查询信号处理函数被触发的次数。

manu@manu-hacks:~/code/c/self/signal$ ./signal\_receiver 30 &

[1] 27639

manu@manu-hacks:~/code/c/self/signal$ ./signal\_receiver:PID is 27639

signal for signo(32) failed (Invalid argument)

signal for signo(33) failed (Invalid argument)

I will sleep 30 second

manu@manu-hacks:~/code/c/self/signal$ ./signal\_sender 27639 10 10000

pid = 27639,signo = 10,times = 10000

done

manu@manu-hacks:~/code/c/self/signal$ ./signal\_sender 27639 36 10000

pid = 27639,signo = 36,times = 10000

done

manu@manu-hacks:~/code/c/self/signal$

manu@manu-hacks:~/code/c/self/signal$ signo(10) :User defined signal 1

signo(36) :Real-time signal 2

manu@manu-hacks:~/code/c/self/signal$ ./signal\_sender 27639 2 1

pid = 27639,signo = 2,times = 1

done

./signal\_receiver:signal 10 caught 1 time

./signal\_receiver:signal 36 caught 10000 times

[1]+ Done ./signal\_receiver 30

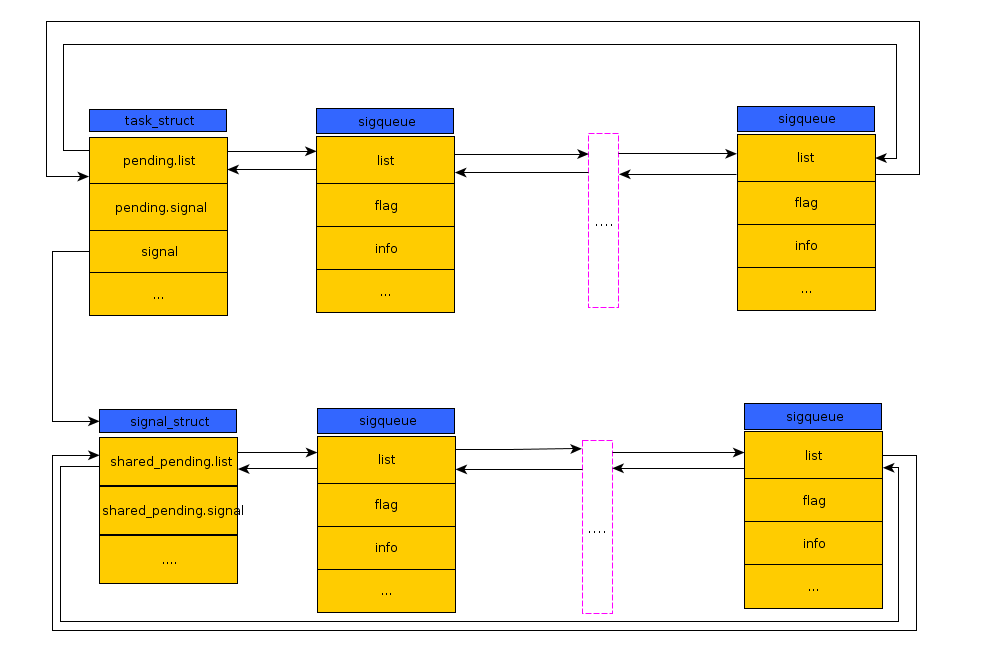
从上面的例子可以看出，如果进程讲信号屏蔽，在此期间，虽然向目标进程发送SIGUSR2 10000次，但是，在解除屏蔽之后，信号处理函数只触发了一次。这就是前面提到的传统信号的不可靠的特性。

其他进程可能会向目标进程发送信号，但是无法确认信号处理函数是否会被执行，原因何在？原因在于传统的信号采用的是位图的方式存储的，对于某信号如SIGUSR2，内核会记录是否存在挂起信号，如果新到来的信号已经存在，处于未决状态，该信号就会被丢弃。

那么实时信号即可靠信号它的表现如何呢？我们发送实时信号36，共计10000次，发现解除屏蔽后，信号处理函数共触发了10000次，没有信号丢失，所有信号都被deliver给了进程去处理。

内核是如何做到这个的呢？这我们就要看内核的实现了。

我们看下进程描述符中与singal相关的数据结构：



进程描述符中存在一个成员变量称之为pending，顾名思义，就是记录处于挂起状态的信号，即内核已经接收到但尚未递送给目标进程的信号集合。

struct task\_struct {

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

void \*stack;

atomic\_t usage;

unsigned int flags; /\* per process flags, defined below \*/

unsigned int ptrace;

...

...

/\* signal handlers \*/

struct signal\_struct \*signal;

struct sighand\_struct \*sighand;

sigset\_t blocked, real\_blocked;

sigset\_t saved\_sigmask;

struct sigpending pending;//用来保存进程的挂起信号

...

}

struct sigpending {

struct list\_head list;

sigset\_t signal;

};

#define \_NSIG 64

#ifdef \_\_i386\_\_

# define \_NSIG\_BPW 32

#else

# define \_NSIG\_BPW 64

#endif

#define \_NSIG\_WORDS (\_NSIG / \_NSIG\_BPW)

typedef unsigned long old\_sigset\_t; /\* at least 32 bits \*/

typedef struct {

unsigned long sig[\_NSIG\_WORDS];

} sigset\_t;

我们可以看下struct sigpending的定义，一个链表头，一个sigset\_t类型的signal，这个sigset\_t类型的变量，就是前面提到的位图。如果在32位系统上，unsigned long 是32 bit，所以需要2个unsigned long才能凑齐64个bit，存储64个不同信号的有无，如果64位系统，只需要一个unsigned long就能凑齐所需要的64 bit，用来表征对应信号值信号的有无。

bitmap无法记录收到信号的次数，这个是比较容易理解的，那么实时信号又是如何做到不丢失信号的呢？答案很简单：链表。sigpending结构体中存在成员变量list，这是链表头，对于实时信号，内核会分配一个struct sigqueue类型的变量，链入到list对应的链表之中，这样就确保信号不会丢失，从而保证信号的可靠性。

struct sigqueue {

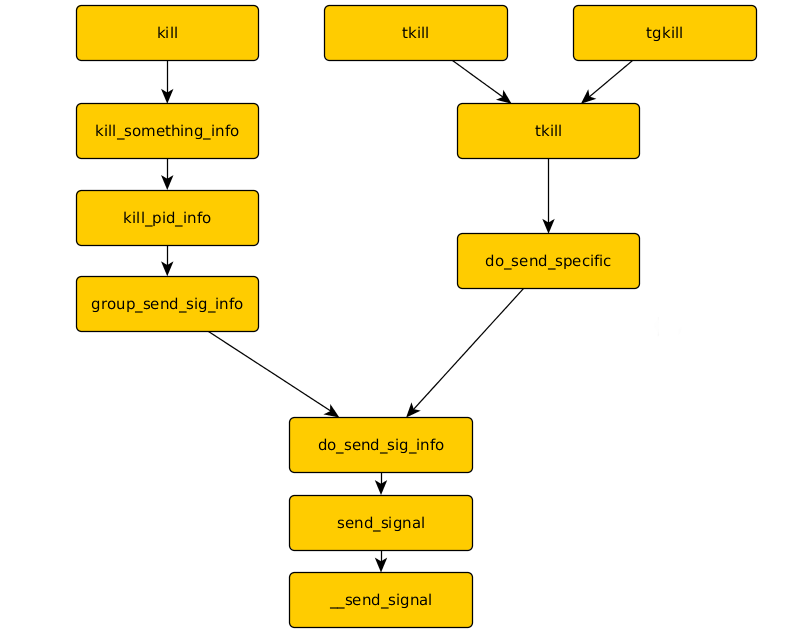
struct list\_head list;

int flags;

siginfo\_t info;

struct user\_struct \*user;

};



从内核代码来看我们常用的kill，还有向线程发送信号的tkill和tgkill，最终都会汇聚到同一个函数，\_\_send\_signal，如上图所示（当然，kill根据pid的值不同，代码调用流程也不同，上图给出的是最基本的pid > 0的情况）。下面我们通过学习\_\_send\_signal来研究，内核如何通过位图和链表，实现非可靠信号的不可靠，以及实时信号的可靠。

static int \_\_send\_signal(int sig, struct siginfo \*info,

struct task\_struct \*t,int group,

int from\_ancestor\_ns)

{

struct sigpending \*pending;

struct sigqueue \*q;

int override\_rlimit;

int ret = 0, result;

assert\_spin\_locked(&t->sighand->siglock);

result = TRACE\_SIGNAL\_IGNORED;

if (!prepare\_signal(sig, t,

from\_ancestor\_ns || (info == SEND\_SIG\_FORCED)))

goto ret;

**pending = group ? &t->signal->shared\_pending : &t->pending;**

/\*

\* Short-circuit ignored signals and support queuing

\* exactly one non-rt signal, so that we can get more

\* detailed information about the cause of the signal.

\*/

result = TRACE\_SIGNAL\_ALREADY\_PENDING;

//如果是低于32的信号，并且已经在pending中出现了的该信号，就直接返回

**if (legacy\_queue(pending, sig))**

goto ret;

result = TRACE\_SIGNAL\_DELIVERED;

/\*

\* fast-pathed signals for kernel-internal things like SIGSTOP

\* or SIGKILL.

\*/

if (info == SEND\_SIG\_FORCED)

goto out\_set;

if (sig < SIGRTMIN)

override\_rlimit = (is\_si\_special(info) || info->si\_code >= 0);

else

override\_rlimit = 0;

//分配sigqueue结构，并且链入到相应的pending。

**q = \_\_sigqueue\_alloc(sig, t, GFP\_ATOMIC | \_\_GFP\_NOTRACK\_FALSE\_POSITIVE,override\_rlimit);**

if (q) {

list\_add\_tail(&q->list, &pending->list);

switch ((unsigned long) info) {

case (unsigned long) SEND\_SIG\_NOINFO:

q->info.si\_signo = sig;

q->info.si\_errno = 0;

q->info.si\_code = SI\_USER;

q->info.si\_pid = task\_tgid\_nr\_ns(current,

task\_active\_pid\_ns(t));

q->info.si\_uid = from\_kuid\_munged(current\_user\_ns(), current\_uid());

break;

case (unsigned long) SEND\_SIG\_PRIV:

q->info.si\_signo = sig;

q->info.si\_errno = 0;

q->info.si\_code = SI\_KERNEL;

q->info.si\_pid = 0;

q->info.si\_uid = 0;

break;

default:

copy\_siginfo(&q->info, info);

if (from\_ancestor\_ns)

q->info.si\_pid = 0;

break;

}

userns\_fixup\_signal\_uid(&q->info, t);

} else if (!is\_si\_special(info)) {

if (sig >= SIGRTMIN && info->si\_code != SI\_USER) {

/\*

\* Queue overflow, abort. We may abort if the

\* signal was rt and sent by user using something

\* other than kill().

\*/

result = TRACE\_SIGNAL\_OVERFLOW\_FAIL;

ret = -EAGAIN;

goto ret;

} else {

/\*

\* This is a silent loss of information. We still

\* send the signal, but the \*info bits are lost.

\*/

result = TRACE\_SIGNAL\_LOSE\_INFO;

}

}

out\_set:

signalfd\_notify(t, sig);

sigaddset(&pending->signal, sig); //加入位图

complete\_signal(sig, t, group);

ret:

trace\_signal\_generate(sig, info, t, group, result);

return ret;

}

static inline int legacy\_queue(struct sigpending \*signals, int sig)

{

//是不可靠信号，并且该信号已经存在挂起信号

return (sig < SIGRTMIN) && sigismember(&signals->signal, sig);

}

static inline int sigismember(sigset\_t \*set, int \_sig)

{

unsigned long sig = \_sig - 1;

if (\_NSIG\_WORDS == 1)

return 1 & (set->sig[0] >> sig);

else

return 1 & (set->sig[sig / \_NSIG\_BPW] >> (sig % \_NSIG\_BPW));

}

关键的代码在于加粗的三行，第二行判断是传统的不可靠信号，并且通过查找位图，发现该信号已经存在挂起信号，无需做过多的处理，只需要返回即可。这种逻辑造就了传统信号的非可靠性。当然了如果传统非实时信号尚未有挂起信号，也是需要调用\_\_sigqueue\_alloc分配sigqueue，链入pending对应的链表。

对于实时信号，纵然已经存在挂起信号，仍然会调用\_\_sigqueue\_alloc来分配sigqueue结构体，链入pending对应的链表。

那么实时信号是不是可以无限制地挂入队列呢？也不是，实际上，内核也做了限制，一个进程，默认挂起信号的个数是有限的，默认是15408，超过这个数，可靠信号变得也没那么可靠，也会丢失信号。我们看下内核代码:

static struct sigqueue \*

\_\_sigqueue\_alloc(int sig, struct task\_struct \*t,

gfp\_t flags, int override\_rlimit)

{

struct sigqueue \*q = NULL;

struct user\_struct \*user;

……

rcu\_read\_lock();

user = get\_uid(\_\_task\_cred(t)->user);

atomic\_inc(&user->sigpending);

rcu\_read\_unlock();

if (override\_rlimit ||

**atomic\_read(&user->sigpending) <=**

**task\_rlimit(t, RLIMIT\_SIGPENDING)**) {

q = kmem\_cache\_alloc(sigqueue\_cachep, flags);

} else {

print\_dropped\_signal(sig);

}

if (unlikely(q == NULL)) {

atomic\_dec(&user->sigpending);

free\_uid(user);

} else {

INIT\_LIST\_HEAD(&q->list);

q->flags = 0;

q->user = user;

}

return q;

}

加粗部分的逻辑，决定了实时信号也不能无限制的挂起，进程挂起信号超过进程的资源限制之后，就会放弃分配，直接丢失该信号。

可以通过如下命令查看系统的限制：

manu@manu-hacks:~$ ulimit -a

core file size (blocks, -c) 0

data seg size (kbytes, -d) unlimited

scheduling priority (-e) 0

file size (blocks, -f) unlimited

pending signals (-i) 15408

....

用上面的测试程序测试一下，看下实时信号是否也会丢失信号：

manu@manu-hacks:~/code/c/self/signal$ ./signal\_receiver 30 &

[1] 14699

manu@manu-hacks:~/code/c/self/signal$ ./signal\_receiver:PID is 14699

signal for signo(32) failed (Invalid argument)

signal for signo(33) failed (Invalid argument)

I will sleep 30 second

manu@manu-hacks:~/code/c/self/signal$ ./signal\_sender 14699 36 20000

pid = 14699,signo = 36,times = 20000

done

manu@manu-hacks:~/code/c/self/signal$ ./signal\_sender 14699 2 1

pid = 14699,signo = 2,times = 1

done

manu@manu-hacks:~/code/c/self/signal$

manu@manu-hacks:~/code/c/self/signal$ signo(2) :Interrupt

signo(36) :Real-time signal 2

./signal\_receiver:signal 36 caught **15408** times

[1]+ Done ./signal\_receiver 30

果然和预期的一样，发送了实时信号36共计20000次，但是信号deliver了15408次，超出部分都被丢弃掉了，如同非实时信号一样。

这个挂起信号的上限值是可以修改的，可以用ulimit -i unlimited这个命令将进程挂起信号的最大值设为无穷大，从而确保内核不会主动丢弃实时信号。注意的是，需要有root的权限。

root@manu-hacks:~/code/c/self/signal# ulimit -i unlimited

root@manu-hacks:~/code/c/self/signal# ./signal\_receiver 30 &

[1] 14820

root@manu-hacks:~/code/c/self/signal# ./signal\_receiver:PID is 14820

signal for signo(32) failed (Invalid argument)

signal for signo(33) failed (Invalid argument)

I will sleep 30 second

root@manu-hacks:~/code/c/self/signal# ./signal\_sender 14820 36 20000

pid = 14820,signo = 36,times = 20000

done

root@manu-hacks:~/code/c/self/signal# ./signal\_sender 14820 2 1

pid = 14820,signo = 2,times = 1

done

root@manu-hacks:~/code/c/self/signal#

root@manu-hacks:~/code/c/self/signal# signo(2) :Interrupt

signo(36) :Real-time signal 2

./signal\_receiver:signal 36 caught 20000 times

[1]+ Done ./signal\_receiver 30

本节用实验的方法证实了非可靠信号可能会丢失，而实时信号基本不会丢失，每向进程发送一次实时信号，基本可以确保信号处理函数会被执行一次。然后从内核层面解释了两者表现不同的原因

## 6.5 信号的安装

前面讲了很多传统信号的弊端，讲了signal的兼容性问题，有问题就会有解决方案。Linux在发展的过程中，提供了新的信号安装方法：sigaction函数。我们不再考虑那些模棱两可API，完全定制自己需要的信号处理函数。

#include <signal.h>

int sigaction(int signum, const struct sigaction \*act,

struct sigaction \*oldact);

struct sigaction {

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

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

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

这个sigaction结构体，不是新朋友了，前面介绍rt\_sigaction的时候，已经提到了sigaction结构体。

sa\_mask顾名思义，就是信号处理函数执行期间，屏蔽信号集。前文介绍bsd\_signal的时候，我们知道，如果我们注册SIGINT信号，内核会自动将SIGINT添加入屏蔽信号集，在信号执行期间，SIGINT信号不会被deliver给进程。但是，也仅仅是SIGINT，如果执行SIGINT期间，想屏蔽SIGHUP，SIGUSR1，那么bsd\_signal就爱莫能助了。但是对sigaction函数而言，根本就不是问题：

struct sigaction sa；

sa.sa\_mask = SIGHUP|SIGUSR1|SIGINT;

需要提到的是，并不是所有的信号都能被屏蔽，SIGKILL和SIGSTOP，不可以为它们注册信号处理函数，也不能够屏蔽这些信号，原因是，系统总要控制某些进程，如果进程可以自行设计所有信号的处理函数，操作系统可能无法控制这些进程。换言之，操作系统是终极boss，需要杀死某些进程的时候，必须可以做到，SIGKILL和SIGSTOP不能被屏蔽，就是防止出现进程无法无天而操作系统徒叹奈何的困境。

如果强制为SIGKILL和SIGSTOP注册信号处理函数，会返回ERROR，并置errno为EINVAL。感兴趣可以自己测试下。

比较有意思的是sa\_flags。前面我们已经介绍了一些标志位，下面我们简要介绍sa\_flags的含义。

**SA\_NOCLDSTOP**

这个标志位只用于SIGCHLD信号。进程管理一章曾经提到，父进程可以监测子进程的三种事件：

* 子进程终止（即子进程死亡）
* 子进程停止（即子进程暂停）
* 子进程恢复（即子进程从暂停中恢复执行）

其中SA\_NOCLDSTOP标志位时用来控制第二和第三事件的。即一旦父进程为SIGCHLD信号设置了这个标志位，那么子进程停止和子进程恢复这两件事情，就无需向父进程发送SIGCHLD信号了。

**SA\_NOCLDWAIT**

这个标志只用于SIGCHLD信号。这个标志位时用来控制上面提到的子进程终止的行为。如果父进程为SIGCHLD设置了SA\_NOCLDWAIT标志位，那么子进程退出时，就不会进入僵尸状态。但是会不会向父进程发送SIGCHLD信号呢？这取决于实现。对于Linux而言，仍然会发送SIGCHLD信号。这点和上面的SA\_NOCLDSTOP略有不同。

**SA\_ONESHOT**

**SA\_RESETHAND**

这两个信号本质是一样的，表示信号处理函数是一次性的，信号递送出去之后，信号处理函数恢复成默认值SIG\_DFL。

**SA\_NODEFER**

**SA\_NOMASK**

这两个信号的本质是一样的，在信号处理函数执行期间，不阻塞当前信号。

**SA\_RESTART**

这个标志位置位表示，如果系统调用被信号中断，不返回错误，而是重启系统调用。

**SA\_SIGINFO**

这个信号表示，信号发送者会提供额外的信息，一般信号处理函数为：

void handle(int, siginfo\_t \*, void \*);

此处重点讲述下带SA\_SIGINFO标志为的信号安装方式。本章序言提到过，signal的本质是一种进程间的通信。一个进程向另外一个进程发送信号，能够传递的信息，不仅仅是signo，还可发送更多的信息，而接收进程也能获得出发送进程的PID，UID，以及发送的额外信息。

我们看下面的例子：

#include<stdio.h>

#include<stdlib.h>

#include<signal.h>

void sig\_int(int signo,siginfo\_t \*info,void \*context)

{

printf("\nget signal:%d\n",signo);

printf("signal number is %d\n",info->si\_signo);

printf("pid=%d\n",info->si\_pid);

printf("sigval = %d\n",info->si\_value.sival\_int);

}

int main(void)

{

struct sigaction new\_action;

sigemptyset(&new\_action.sa\_mask);

new\_action.sa\_sigaction = sig\_int;

new\_action.sa\_flags |= SA\_SIGINFO|SA\_RESTART;

if(sigaction(36,&new\_action,NULL)==-1){

printf("set signal process mode\n");

exit(1);

}

while(1)

pause();

printf("Done\n");

exit(0);

}

这个例子比较简单，我们为36号信号注册了信号处理函数，信号处理函数执行期间，屏蔽SIGINT信号，如果信号中断了系统调用，内核重启系统调用。我们带上了SA\_SIGINFO，所以我们的信号处理函数采用了

void sig\_int(int signo,siginfo\_t \*info,void \*context)

这个函数也很有意思，info->si\_pid信号发送者的PID，info->si\_value.sival\_int是信号发送进程发送的int值。发送进程和接收进程约定好，发送者使用sigqueue发送信号，同时带上int型的额外信息，接收进程就能获得发送进程的PID以及整型的额外信息。

那么系统说来，如果带了SIGINFO标志位，进程可以获得哪些信息？下一节介绍sigqueue函数时，会展开讲述。

## 6.6 信号的发送

要想触发信号处理函数，必然要有信号的发送。没有发送者，信号处理函数只是摆设而已。

有很多触发信号的方式：

**硬件异常产生信号**

硬件异常，可能产生SIGBUS、SIGFPE、SIGILL和SIGSEGV信号。当然也可以调用kill函数来产生这些信号，但一般不会这样用。

SIGBUS一般是由于访问了非法的地址。SIGBUS和SIGSEGV有点类似。但是略有区别。SIGSEGV多发生在地址是合法有效的，但是没有相应的权限。比如用户程序访问地址NULL或者访问地址0；而SIGBUS的关注点在地址不合法。一般来讲，地址不对齐，是触发SIGBUS的一种常见场景。比如int型变量一般是4字节对齐的，如果传入的地址不能被4整除，将会触发SIGBUS信号。此外，mmap函数中，将文件映射到内存，如果访问超出文件长度的地址，将会引发SIGBUS信号。

SIGFPE信号，一般由算术异常引发。表面看SIGFPE是负责浮点数异常的，实际上，覆盖的范围不仅仅是浮点数异常。包括除以0，包括溢出。

SIGILL信号，这个SIGILL的含义是illegal instruction，不合法的指令。一般程序尝试执行不存在的指令或者需要特权的指令时才会触发。

SIGSEGV信号，这个绝对是C程序员的梦魇，出现的概率要大于前面几个。一般来说，当程序尝试读或者写程序分配内存以外的地址，或者尝试修改只读内存地址会出现SEGSEGV信号。更常见的触发方法是访问NULL地址或者访问一个未初始化的地址；还有一种触发方法是栈空间遭到破坏，原本合法的地址被篡改。

上面这四种硬件异常，一般是由程序自身引发的，不是有其他进程发送的信号，并且这些异常比较致命，以至于进程无法继续下去。所以这些信号产生后，会立刻递送给进程。默认情况下，这四种信号都会是进程终止，并且产生core dump文件供调试。这些信号，进程不能忽略，而且也不能阻塞该信号。

**终端按键引发终端产生信号**

终端提供了一些发送信号的快捷键，可以通过stty -a查看终端的设置：

speed 38400 baud; rows 35; columns 157; line = 0;

intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;

从上面可看出终端提供了一些快捷键来发送信号给前台进程组的所有进程

* Ctrl+c : 发送SIGINT信号，信号的默认行为是进程终止；
* Ctrl+\ : 发送SIGQUIT信号，信号的默认行为是进程终止，并且产生core dump文件；
* Ctrl+z：发送SIGTSTP信号，是进程暂停，并且转入后台执行。

其中Ctrl+z快捷键称为挂起操作，非常有用，比如正在使用vim写代码，但是你需要跳回到终端执行几个命令，你又不想退出vim，也不愿意再打开其他终端，这时候你可以使用Ctrl+z快捷键，使vim进程暂停，并且转入后台。执行完你想做的事情之后，使用fg命令将暂停的vim进程恢复执行，并且转入前台执行。

注意SIGSTOP和SIGTSTP信号都可以是进程暂停执行，两者的区别在于，SIGSTOP信号不能忽略，不能捕获，不能忽略，不能阻塞，和SIGKILL类似。但是SIGTSTP不同，它只是普通信号，进程可以忽略之，捕获之，阻塞之。

在终端上，可以通过kill -signo PID命令向目标进程发送信号，比如发送SIGHUP信号给进程ID为234的进程，可以通过执行如下命令做到：

kill -2 234 或者

kill -SIGHUP 234

**软件条件触发信号**

这也是一类产生信号的方法。比较典型的有子进程终止，会向父进程发送SIGCHLD信号。当管道的所有读取端都关闭了，再向管道写入数据会产生SIGPIPE信号等。

除了上面提到的产生信号的方法，剩下的就是通过调用kill家族函数，向其他进程或者自己发送信号了。

### 6.6.1 kill，tkill和tgkill

kill的使用方式如下：

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig);

根据pid的值，分成一下几种情况

* pid > 0 ,发送信号给PID = pid的进程
* pid = 0 ，发送信号给调用进程同一个进程组的每一个进程
* pid = -1，向调用进程有权限发送信号的进程，发出信号，init进程和进程自身除外
* pid < -1，向进程组-pid 发送信号。

这么多情况，最常见的就是第一种情况，向指定进程发送信号。

如何向线程发送信号？

Linux 提供了tkill和tgkill两个系统调用来向某个线程发送信号。

int tkill(int tid, int sig);

int tgkill(int tgid, int tid, int sig);

这两个是原生态的系统调用，glibc并没有提供对这两个系统调用的封装，所以如果我们想使用这两个函数，需要采用syscall的方式，具体如何调用，可以参考下面的示例代码。

等一下，为什么有了tkill，还要引入tgkill？

tgkill的第一个参数tgid，线程组中主线程的tid，或者称为进程号。看起来是多余的，其实起到一个保护的作用，防止发送信号到错误的线程。进程号或者线程ID这种资源是内核负责管理的，进程有自己的生命周期，比如向线程ID为123的线程发送信号，很可能线程123早就退出了，线程ID 123被内核分配给另一个不相干的进程。这种情况下，直接调用tkill，就可能发送信号到不相干的进程。为了防止出现这种情况，引入了tgkill系统调用，含义是向线程组ID是tgid，线程ID为tid的线程发送信号。这样误杀的概率就大大减少了。

我们用个例子来讲解如何向线程发送信号。

#include<signal.h>

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<sys/syscall.h>

#define NTHREAD 20

\_\_thread int pid;

\_\_thread int tid;

void\* thread\_func()

{

pid = getpid();

tid = syscall(SYS\_gettid);

fprintf(stderr,"I am thread %lx,PID(%d),TID(%d) \n",pthread\_self(),pid,tid);

while(1)

{

pause();

}

}

void signal\_handler(int signo)

{

char MSG[128] = {0};

if(signo == SIGINT)

{

snprintf(MSG,128,"I am PID(%d) TID(%d) I catch the SIGNAL SIGINT\n",pid,tid);

write(1,MSG,sizeof(MSG));

}

}

int main()

{

pthread\_t thread\_id[NTHREAD] ;

int i = 0;

int ret = 0;

struct sigaction sa;

pid = getpid();

tid = syscall(SYS\_gettid);

sa.sa\_handler = signal\_handler;

sa.sa\_flags |= SA\_RESTART;

sigemptyset(&sa.sa\_mask);

ret = sigaction(SIGINT,&sa,NULL) ;

if(ret == -1)

{

fprintf(stderr,"failed to install signal to process\n");

return -1;

}

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

{

ret = pthread\_create(&thread\_id[i],NULL,thread\_func,NULL);

if(ret != 0)

{

fprintf(stderr,"failed to create thread %d\n",i);

return -2;

}

}

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

{

pthread\_join(thread\_id[i],NULL);

}

return 0;

}

上面这个进程，会创建20个线程，在创建线程之前，注册了信号处理函数。主进程的信号处理函数，是否被继承到了线程？如果向主进程发送信号SIGINT，哪个线程会负责执行信号处理函数？如何向20个线程中的指定线程发送信号？这些都是困扰我们的问题。

在本节，我们解决如何向指定的线程发送信号：

#include<stdio.h>

#include<sys/syscall.h>

#include<getopt.h>

#include<string.h>

#include<errno.h>

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

{

unsigned long tid = 0;

unsigned long tgid = 0;

int signo = -1;

int check\_group = 0;

int ret = -1;

static const struct option options[] = {

{"process\_id",required\_argument,NULL,"p"},

{"signal\_no",required\_argument,NULL,"s"},

{"check\_group",required\_argument,NULL,"g"},

{}

};

for(;;)

{

int option ;

option = getopt\_long(argc,argv,"p:s:g:",options,NULL);

if(option == -1)

break;

switch(option)

{

case 'p':

tid = strtol(optarg,NULL,0);

break;

case 's':

signo = atoi(optarg);

break;

case 'g':

check\_group = 1;

tgid = strtol(optarg,NULL,0);

break;

default:

ret = -1;

goto exit;

}

}

if(tid == 0 || signo == -1)

{

ret = -2;

goto exit;

}

if(check\_group == 1)

{

ret = syscall(SYS\_tgkill,tgid,tid,signo);

}

else

{

ret = syscall(SYS\_tkill,tid,signo);

}

if(ret != 0 )

{

fprintf(stderr,"send signo(%d) to tid(%lu) tgid(%lu) failed,reason(%s)\n",signo,tid,tgid,strerror(errno));

goto exit;

}

ret = 0;

exit :

return ret ;

}

在上面的示例代码中我们对tkill和tgkill进行了简单的封装，将系统调用封装成了一个binary tool，我们可以利用这个tool进行测试：

首先我们启动我们的thread\_group，创建20个子线程+自身主线程，共21个线程，我们只在主进程注册了信号处理函数，我们来观察，能否向线程发射信号，线程能否收到信号，线程是否执行主线程的信号处理函数:

manu@manu-hacks:~/code/c/self/signal$ ./thread\_signal

I am thread b6d98b40,PID(6938),TID(6940)

I am thread b7599b40,PID(6938),TID(6939)

I am thread b5595b40,PID(6938),TID(6943)

I am thread b5d96b40,PID(6938),TID(6942)

I am thread b6597b40,PID(6938),TID(6941)

I am thread b4d94b40,PID(6938),TID(6944)

I am thread b4593b40,PID(6938),TID(6945)

I am thread b3d92b40,PID(6938),TID(6946)

I am thread b3591b40,PID(6938),TID(6947)

I am thread b2d90b40,PID(6938),TID(6948)

I am thread b258fb40,PID(6938),TID(6949)

*I am thread b1d8eb40,PID(6938),TID(6950)*

I am thread b158db40,PID(6938),TID(6951)

I am thread b058bb40,PID(6938),TID(6953)

I am thread afd8ab40,PID(6938),TID(6954)

I am thread aed88b40,PID(6938),TID(6956)

I am thread af589b40,PID(6938),TID(6955)

I am thread b0d8cb40,PID(6938),TID(6952)

I am thread add86b40,PID(6938),TID(6958)

I am thread ae587b40,PID(6938),TID(6957)

可以看到创建20个线程完毕，所有线程都拥有相同的进程ID 6938，但是每个都有自己的thread id。下面我们调用我们的tkill tool来向thread id = 6950的线程发送信号SIGINT。在另一个终端上执行：

./tkill -p 6950 -s 2

在刚才的终端上，我们可以看到：

I am PID(6938) TID(6950) I catch the SIGNAL SIGINT

这个简单的例子说明了两点：

1. 主线程注册的信号处理函数，线程组里的线程得到了继承
2. tkill可以准确的发送信号到指定线程

同一个线程组的其他线程，为什么可以继承主线程的信号处理函数。发送信号给多线程的进程，进程会选择哪一个线程来处理信号。多线程程序收到kill信号的时候，或者触发core dump的时候，为什么所有线程都退出了，我们会在后面的线程与信号一节中讲述这些内容。

### 6.6.2 raise

Linux提供了向自身发送信号的接口：raise。

#include <signal.h>

int raise(int sig);

这个接口对于单线程的程序而言，就相当与执行

kill(getpid(),sig)

这个接口对于多线程的程序而言，就相当与执行

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

执行成功的时候，返回0，否则返回非零值。值得注意的是，信号处理函数执行完毕之后，raise才能返回。

### 6.6.3 sigqueue

信号发送方式当中，sigqueue算是后起之秀，传统的信号多用signal/kill这两个函数搭档，完成信号的安装和信号的发送。后来signal函数的表达力不够，使用不够灵活，所以引入了sigaction函数来完成信号的安装，与其对应的，有sigqueue函数来完成实时信号的发送，当然了，sigqueue也能发送非实时信号。

其接口如下

#include <signal.h>

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

其中pid和sig都顾名思义，比较有意思的是第三个入参：

union sigval {

int sival\_int;

void \*sival\_ptr;

};

前面提到过，信号本质是一种进程间的通信，传递的信息不但有信号的值，还可以传递更多的信息，我们可以看到，我们通过sigqueue可以传递一个int值。当然了sigval是一个联合体，还可发送一个指针。但是考虑到，不同进程地址空间，传递指针对于另一个进程几乎没有任何的意义。一些简单的消息传递，完全可以使用sigqueue，通过sigval中的sival\_int来通知接收进程作相应的事情。可惜的是传递的信息终究是有限。

下面的一个例子，使用了sigqueue函数向进程发送信号，目标进程，信号值，发送次数，都可指定，发送信号的同时，在消息体中传递了一个int型的值123。

#include <signal.h>

#include <sys/types.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

void usage()

{

fprintf(stderr,"sigqueue\_send sig pid [times]\n");

}

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

{

pid\_t pid;

int sig;

int times = 0;

union sigval mysigval ;

if(argc < 3)

{

usage();

return -1;

}

pid = atoi(argv[1]);

sig = atoi(argv[2]);

if(argc >= 4)

{

times = atoi(argv[3]);

}

mysigval.sival\_int = 123;

if(sig < 0 || sig >64 ||times < 0)

{

usage();

return -2;

}

int i = 0;

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

{

if(sigqueue(pid,sig,mysigval) != 0)

{

fprintf(stderr,"sigqueue failed (%s)\n",strerror(errno));

return -3;

}

}

return 0;

}

一般来说，sigqueue函数的黄金搭档是sigaction函数。前文我们提过sigaction来安装信号，注册信号处理函数。只要建立时，用到SA\_SIGINFO标志位，我们就可以使用带三个参数的信号处理函数来处理实时信号。

struct sigaction act；

act.sa\_flag |= SA\_SIGINFO;

三个参数的信号处理函数如下：

void handle(int, siginfo\_t \*info, void \*ucontext);

siginfo\_t结构体存在以下成员：

siginfo\_t {

int si\_signo;

int si\_errno;

int si\_code;

int si\_trapno;

pid\_t si\_pid;

uid\_t si\_uid;

union sigval si\_value;

void \*si\_addr

...

}

这个结构体包含很多信息，目标进程可以通过该数据结构，获取到相关的信息：

* si\_signo：信号编号
* si\_code：信号来源，可以通过这个值判断信号的来源

|  |  |
| --- | --- |
| **si\_code** | **信号来源** |
| **SI\_USER** | 调用kill或者raise的用户进程 |
| **SI\_TKILL** | 调用tkill或者tgkill的用户进程 |
| **SI\_QUEUE** | 调用sigqueue函数的用户进程 |
| **SI\_MESGQ** | 消息到达POSIX消息队列 |
| **SI\_KERNEL** | 内核产生的信号 |
| **SI\_ASYNCIO** | 异步IO操作完成 |
| **SI\_TIMER** | POSIX定时器到期 |

除此外，一些特殊的信号会产生一些独特的si\_code，来表达信号产生的根源或来源：

比如如果无效地址对齐引发SIGBUS信号，si\_code会被置为BUS\_ADRALN等。如想或者详情，可以查看glibc的bits/siginfo.h头文件。

* si\_value：这个值就是记录sigqueue函数发送信号时，带的伴随数据。
* si\_pid：对于kill或者sigqueue发送信号，si\_pid记录了信号发送者的进程ID
* si\_uid：对于通过kill或者sigqueue发送信号，si\_uid记录了发送者的真实用户ID。
* si\_addr：仅针对硬件产生的信号SIGBUS、SIGFPE、SIGILL和SIGSEGV设置该字段，该字段表示无效内存的地址（SIGBUS和SIGSEGV）或者导致信号产生的程序指令地址（SIGFPE和SIGILL）。

三参数信号处理函数的第三个参数是void\* 类型的，其实本质是一个ucontext\_t类型的变量。

typedef struct ucontext{

unsigned long int uc\_flags;

struct ucontext \*uc\_link;

stack\_t uc\_stack;

mcontext\_t uc\_mcontext;

\_\_sigset\_t uc\_sigmask;

struct \_libc\_fpstate \_\_fpregs\_mem;

} ucontext\_t;

这个结构体提供了进程上下文的信息，用于描述进程执行信号处理函数之前进程所处的状态。信号处理函数很少会用到这个变量，但是也有很精妙的应用。曾经在stackoverflow中看到利用这个打印出函数堆栈的应用。

对于写C程序的程序员而言，基本每个人都会遇到段错误。常在河边走，哪能不湿鞋，常在江湖漂，哪能不挨刀。一般而言，段错误是因为程序访问了非法的内存地址，这种情况下操作系统会发送一个 SIGSEGV信号给进程，导致进程产生核心转储文件并且退出。如何让进程捕捉到SIGSEGV，打印出有用的方便定位问题的信息，优雅的退出？可以给SIGSEGV注册信号处理函数。

#ifndef \_GNU\_SOURCE

#define \_GNU\_SOURCE

#endif

#ifndef \_\_USE\_GNU

#define \_\_USE\_GNU

#endif

#include <execinfo.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ucontext.h>

#include <unistd.h>

typedef struct \_sig\_ucontext {

unsigned long uc\_flags;

struct ucontext \*uc\_link;

stack\_t uc\_stack;

struct sigcontext uc\_mcontext;

sigset\_t uc\_sigmask;

} sig\_ucontext\_t;

void crit\_err\_hdlr(int sig\_num, siginfo\_t \* info, void \* ucontext)

{

void \* array[50];

void \* caller\_address;

char \*\* messages;

int size, i;

sig\_ucontext\_t \* uc;

uc = (sig\_ucontext\_t \*)ucontext;

**caller\_address = (void \*) uc->uc\_mcontext.eip**;

fprintf(stderr, "signal %d (%s), address is %p from %p\n",

sig\_num, strsignal(sig\_num), info->si\_addr,

(void \*)caller\_address);

size = backtrace(array, 50);

/\* overwrite sigaction with caller's address \*/

array[1] = caller\_address;

messages = backtrace\_symbols(array, size);

/\* skip first stack frame (points here) \*/

for (i = 1; i < size && messages != NULL; ++i)

{

fprintf(stderr, "[bt]: (%d) %s\n", i, messages[i]);

}

free(messages);

exit(EXIT\_FAILURE);

}

int crash()

{

char \* p = NULL;

\*p = 0;

return 0;

}

int foo4()

{

crash();

return 0;

}

int foo3()

{

foo4();

return 0;

}

int foo2()

{

foo3();

return 0;

}

int foo1()

{

foo2();

return 0;

}

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

{

struct sigaction sigact;

sigact.sa\_sigaction = crit\_err\_hdlr;

sigact.sa\_flags = SA\_RESTART | SA\_SIGINFO;

if (sigaction(SIGSEGV, &sigact, (struct sigaction \*)NULL) != 0)

{

fprintf(stderr, "error setting signal handler for %d (%s)",

SIGSEGV, strsignal(SIGSEGV));

exit(EXIT\_FAILURE);

}

foo1();

exit(EXIT\_SUCCESS);

}

上面函数利用的第三个参数里面的ucontext->uc\_mcontext.eip字段，获取到了收到信号前的EIP寄存器的值，根据该值，可以讲堆栈信息打印出来。

root@manu-hacks:~/code/c/self/signal# gcc -o print\_bt print\_core.c

root@manu-hacks:~/code/c/self/signal# ./print\_bt

signal 11 (Segmentation fault), address is (nil) from 0x80486c2

[bt]: (1) ./print\_bt() [0x80486c2]

[bt]: (2) ./print\_bt() [0x80486c2]

[bt]: (3) ./print\_bt() [0x80486d4]

[bt]: (4) ./print\_bt() [0x80486e3]

[bt]: (5) ./print\_bt() [0x80486f2]

[bt]: (6) ./print\_bt() [0x8048701]

[bt]: (7) ./print\_bt() [0x8048786]

[bt]: (8) /lib/i386-linux-gnu/libc.so.6(\_\_libc\_start\_main+0xf5) [0xb75fa935]

[bt]: (9) ./print\_bt() [0x80484b1]

缺点是没有打印出函数，只打印了指令的地址，我们虽然可以根据objdump出汇编文件，去查找这些指令的地址，从而确定函数，但是手工干预太多。编译的时候，带上-rdynamic选项，我们就可打印出函数的地址了。

root@manu-hacks:~/code/c/self/signal# gcc -o print\_bt print\_core.c -rdynamic

root@manu-hacks:~/code/c/self/signal# ./print\_bt

signal 11 (Segmentation fault), address is (nil) from 0x8048992

[bt]: (1) ./print\_bt(crash+0x10) [0x8048992]

[bt]: (2) ./print\_bt(crash+0x10) [0x8048992]

[bt]: (3) ./print\_bt(foo4+0x8) [0x80489a4]

[bt]: (4) ./print\_bt(foo3+0x8) [0x80489b3]

[bt]: (5) ./print\_bt(foo2+0x8) [0x80489c2]

[bt]: (6) ./print\_bt(foo1+0x8) [0x80489d1]

[bt]: (7) ./print\_bt(main+0x7e) [0x8048a56]

[bt]: (8) /lib/i386-linux-gnu/libc.so.6(\_\_libc\_start\_main+0xf5) [0xb75c7935]

[bt]: (9) ./print\_bt() [0x8048781]

可以很清楚的看到堆栈调用关系，方便我们定位问题。

## 6.7 信号与线程的关系

前面也曾简单提到多线程，比如如何向多线程中的某个发送信号，本节重点讲述多线程与信号的关系。

提到线程与信号的关系，必须提POSIX标准，Linux也遵循POSIX标准：

* 信号处理函数必须在多线程应用的所有线程之间共享，但是，每个线程要有自己的挂起信号掩码和阻塞信号掩码。
* POSIX 函数kill/sigqueue必须面向所有的多线程应用而不是某个特殊的线程
* 每个发给多线程应用的信号仅传送给1个线程，这个线程是由内核从不会阻塞该信号的线程中随意选出
* 如果发送一个致命信号到多线程，那么内核将杀死该应用的所有线程，而不仅仅是接收信号的那个线程。

这些是POSIX标准的要求，Linux要遵循这些要求，Linux是怎么做到的呢？

在进程管理一章，曾经提到过内核的进程描述符的一些成员变量

struct task\_struct {

pid\_t pid;

pid\_t tgid

.....

struct task\_struct \*group\_leader; /\* threadgroup leader \*/

......

struct list\_head thread\_group;

....

}

线程描述符有两个pid\_t类型的变量，从字面上看，pid是进程的进程id，其则不然，pid本质是进程的线程ID，而tgid是thread group id，才是真正的进程ID。

如何理解？

对于多线程程序而言，无论哪个线程，执行getpid，返回值都是一样的，最终返回的都是内核中进程描述符的tgid字段。但是如果执行gettid系统调用，就会发现各个线程返回的值是不一样的，而这个值对应的是进程描述符的pid字段。

单线程的程序，线程ID （即进程描述符中的pid）等于线程组ID（即进程描述符中的tgid），group\_leader指向自己的进程描述符。对于多线程程序，第一个主线程，内核会分配线程ID，线程组ID等于自身的线程ID。group\_leader指向自身的进程描述符。主线程可能会通过pthread\_create来创建其他线程。这时候，CLONE\_THREAD标志位等于1，来告知内核，这是主线程创建的子线程。此时，内核会分配新的线程ID（即进程描述符中的pid），但是线程组ID指向创建者即主线程的线程组ID。同时group\_leader指向创建者即主线程的进程描述符，并且将线程通过链表头，链在一起，方便遍历。

### 6.7.1共享信号处理函数和私有阻塞信号掩码

我们都知道，无论线程创建还是进程创建，最终都会走到do\_fork，到底是创建线程还是进程传进来的标志位来决定的。而do\_fork会调用copy\_process来完成大部分的工作。

copy\_process中有如下的代码：

p->pid = pid\_nr(pid);

p->tgid = p->pid;

if (clone\_flags & CLONE\_THREAD)//创建线程，tgid等于当前调用线程的

p->tgid = current->tgid;

p->group\_leader = p;//主线程的group\_leader指向自身。

INIT\_LIST\_HEAD(&p->thread\_group);

if (clone\_flags & CLONE\_THREAD) {

//线程处理部分，group\_leader指向主线程的进程描述符。同时挂入链表中

current->signal->nr\_threads++;

atomic\_inc(&current->signal->live);

atomic\_inc(&current->signal->sigcnt);

p->group\_leader = current->group\_leader;

list\_add\_tail\_rcu(&p->thread\_group,

&p->group\_leader->thread\_group);

}

了解了这些，我们可以看下进程描述符中信号相关的数据结构了。

/\* signal handlers \*/

struct signal\_struct \*signal;

struct sighand\_struct \*sighand;

sigset\_t blocked, real\_blocked;

sigset\_t saved\_sigmask;

struct sigpending pending;

上面这些都是进程描述符的成员变量，其中同一线程组的所有线程的signal指针都指向同一个signal\_struct 类型的变量，sighand成员变量也是如此，同一线程组的所有线程指向同一个sighand\_struct类型的变量。

static int copy\_signal(unsigned long clone\_flags,

struct task\_struct \*tsk)

{

struct signal\_struct \*sig;

//线程，直接返回，表明同一线程组共享signal\_struct类型的变量

if (clone\_flags & CLONE\_THREAD)

return 0;

//主线程，则需要分配一个signal\_struct类型的变量

sig = kmem\_cache\_zalloc(signal\_cachep, GFP\_KERNEL);

tsk->signal = sig;

if (!sig)

return -ENOMEM;

sig->nr\_threads = 1;

atomic\_set(&sig->live, 1);

atomic\_set(&sig->sigcnt, 1);

init\_waitqueue\_head(&sig->wait\_chldexit);

sig->curr\_target = tsk;

...

}

static int copy\_sighand(unsigned long clone\_flags,

struct task\_struct \*tsk)

{

struct sighand\_struct \*sig;

if (clone\_flags & CLONE\_SIGHAND) {

//如果发现是线程，直接讲引用计数++，无需分配sighand\_struct结构

atomic\_inc(&current->sighand->count);

return 0;

}

sig = kmem\_cache\_alloc(sighand\_cachep, GFP\_KERNEL);

rcu\_assign\_pointer(tsk->sighand, sig);

if (!sig)

return -ENOMEM;

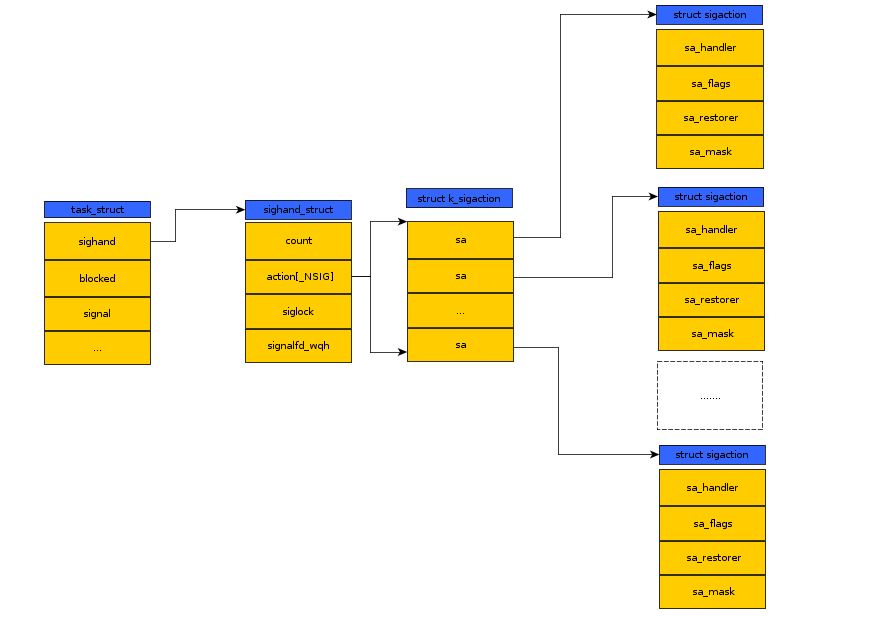
atomic\_set(&sig->count, 1);

memcpy(sig->action, current->sighand->action, sizeof(sig->action));

return 0;

}

内核通过共用一套信号处理函数相关的内核变量sighand\_struct来共享了信号处理函数，多线程程序，安装的信号处理函数，对每个进程而言是一样的，因为他们原本就共用同一个结构体。



我们无论用signal函数还是用sigaction函数安装信号，我们定义的信号处理函数会被记录在内核的sa\_handler。而sa\_mask表示该信号处理函数执行期间，需要屏蔽的信号，sa\_flags前面也提到了。这三个量就完整表达了信号的行为。而多线程中，这64个信号的信号处理行为是共享的。

上图中，进程描述符里的blocked是进程的阻塞信号集，它的本质是一个位图。本线程屏蔽的信号置1。多线程程序，每个线程都有自己的blocked集合。各线程独立的阻塞信号集blocked，符合了POSIX的标准。

glibc提供了sigprocmask函数来指定信号的屏蔽集合。所谓屏蔽，指的是暂时不会将该信号递送给进程，但是会将信号记录在挂起信号之中。一旦进程解除了对该信号的屏蔽，该信号就可以递送给进程，从而触发信号处理函数了。

#include <signal.h>

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

setprocmask函数的行为取决于第一个参数how：

* SIG\_SETMASK：将进程的信号掩码设为set
* SIG\_BLOCK：将set集合中的信号加入到屏蔽信号集合中去。
* SIG\_UNBLOCK：从进程的屏蔽信号集合中移除set中出现的信号。

对于信号集合，sigset\_t，Linux提供了一些函数来组建信号集合：

#include <signal.h>

int sigemptyset (sigset\_t \*set);

int sigfillset (sigset\_t \*set);

int sigaddset (sigset\_t \*set, int signo);

int sigdelset (sigset\_t \*set, int signo);

int sigismember (const sigset\_t \*set, int signo);

sigemptyset()初始化有信号集合，将信号集合设为空，sigfillset()正相反，讲所有信号放置到信号集合之中，一般初始化信号集合会调用这两个函数之一。

sigaddset()将signo放到信号集合set之中，sigdelset()将signo从信号集合set中移除。成功返回0，失败返回-1。一般失败的原因是signo是无效值。

sigismember()用来判断signo是否是信号集合set中成员。signo在信号集合set中，则返回1，signo不在信号集合set之中返回0，出错返回-1（如信号值signo是无效值）。

### 6.7.2私有挂起信号和共享挂起信号

前面我们讲述传统信号的不可靠性的时候，提到了一个关键的数据结构struct sigpengding，我们重点讲解了这个结构体，解释了传统信号为何不可靠，以及实时信号为何可以做到不丢弃信号。

很有意思的是，内核的进程描述符中存在两个struct sigpengding结构体：

struct sigpending pending;

struct signal\_struct \*signal;

在成员变量singal结构体中，还有一个shared\_pending字段：

struct signal\_struct {

atomic\_t sigcnt;

atomic\_t live;

.....

/\* shared signal handling: \*/

struct sigpending shared\_pending;

....

}

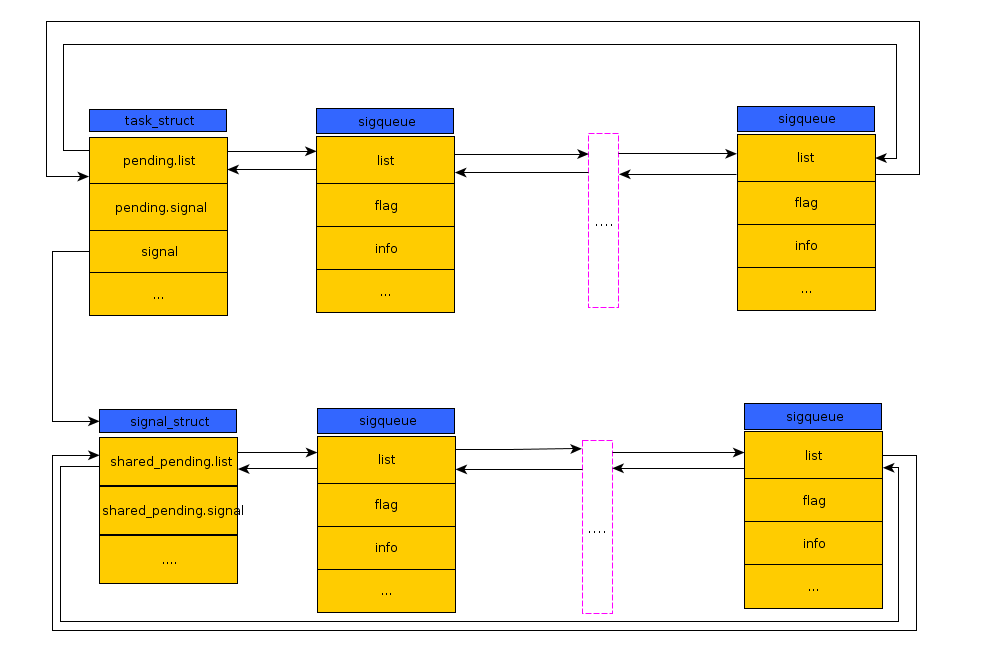
POSIX标准的第二条，就是kill和sigqueue发送的信号必须面对所有进程，而不是某个进程，内核如何做到呢？很简单，shared\_pending是线程组所有线程共享的，pending是各个线程各自独立的。对于不区分线程，向线程组所有线程发送信号的kill和sigqueue，将信号记录在shared\_pending内。对于区分线程，向特定线程发送信号的tkill或者tgkill，内核将信号记录在线程自己的pending内。

内核\_\_send\_signal函数中存在下面一行代码：

pending = group ? &t->signal->shared\_pending : &t->pending;

如果用户调用的是kill或者sigqueue，那么group是1，如果用户调用的是tkill或者tgkill，group参数是0。内核以此来区分该信号是发给线程组的还是发给单个特定线程的。

通过进程描述符中的pending，我们实现了每个线程都有自己的挂起信号集合。通过进程描述符中的signal.shared\_pending ，我们做到了kill/sigqueue发送的信号，对线程组的每个线程可见。从而符合了POSIX标准的第一和第二条。



因为多个线程都共享一个shared\_pending ，所以kill或者sigqueue发送的信号，内核可能从挂起信号中摘下该信号，然后递送给线程组的任意线程。这部分逻辑，我们会在signal的deliver顺序一节中详细讲述。

glibc提供了一个API来获取当前线程的待处理信号。但是待处理信号不是简单的所有挂起信号，即不是私有pending和共享shared\_pending的并集，而是挂起信号中被阻塞的那些信号。只有改变阻塞信号掩码，这些信号才有机会deliver给进程。

我们看下内核代码：

long do\_sigpending(void \_\_user \*set, unsigned long sigsetsize)

{

long error = -EINVAL;

sigset\_t pending;

if (sigsetsize > sizeof(sigset\_t))

goto out;

/\*多线程共享shared\_pending，所以要加锁\*/

spin\_lock\_irq(&current->sighand->siglock);

sigorsets(&pending, &current->pending.signal,

&current->signal->shared\_pending.signal);

spin\_unlock\_irq(&current->sighand->siglock);

/\* Outside the lock because only this thread touches it. \*/

/\*取当前挂起信号的集合，与阻塞信号掩码取交集。\*/

**sigandsets(&pending, &current->blocked, &pending);**

error = -EFAULT;

if (!copy\_to\_user(set, &pending, sigsetsize))

error = 0;

out:

return error;

}

### 6.7.3 致命信号下，进程组全体退出

至于多线程程序的退出，我们知道，有些情况下，我们希望整个线程组的所有线程都退出。引入线程组的概念之后，Linux多了一个exit\_group的系统调用，来完成这件事情。第四章讲述exit的时候，已经分析了这个系统调用，exit\_group基本是调用了do\_exit\_group,完成退出。对于收到致命信号，POSIX标准要求线程组所有线程一并退出。在get\_signal\_to\_deliver中，如果发现信号是致命信号，会调用do\_exit\_group完成线程组的整体退出，当然了，有些信号可能需要产生core文件，在退出之前，会通过do\_coredump产生core dump。

## 6.8等待信号

有时候，我们需要等待信号的发生。POSIX提供了三种机制用来挂起进程，等待信号的递送。分别是pause，sigsuspend，sigwait。

### 6.8.1 pause函数

pause函数将调用线程挂起，直到传递了一个信号为止。这个信号的动作或者是执行了用户定义的信号处理函数，或者是终止进程。如果是执行用户自定义的信号处理函数，pause会在信号处理函数执行完毕后返回，如果是终止进程，pasue函数就不返回了。比较有意思的是pause函数总是返回-1，所以不要误认为返回0。

#include <unistd.h>

int pause (void);

如果我们希望pause函数来等待某个特定的信号，就必须确定哪个信号会让pause返回。事实上，pause并不能主动区分使pause返回的信号是否是我们期待的信号。我们必须间接完成这个任务。常用的方法是，在我们期待的特定信号的信号处理函数中，设一个标志符，待pause返回后，查看标志符是否置位，从而判定我们期待的特定信号是否捕获。

static volatile sig\_atomic\_t sig\_received\_flag = 0;

while(sig\_received\_flag == 0)

pause();

如果等待的信号的信号处理函数会将sig\_received\_flag 置为 1，下面的代码就能阻塞，一直到接收到特定的信号为止。

看起来很美好，可是上面的逻辑是有漏洞的。设想在检查sig\_received\_flag == 0之后，调用pause之前，等待的信号传递给了进程，进程就会依然阻塞。也就是说，等待的信号已经到来，但是被进程错失了，在收到任何一个信号之前，进程都无法发现，其实进程等待的信号早就已经收到。

再举一个例子，来描述一下pause的困境。在编程中，在关键代码片段，我们不期望信号打断正常的流程，我们会临时阻塞信号，关键区域完成之后，我们解除信号的阻塞，然后暂停执行直到有信号到达为止：

/\*end of critical region\*/

sigprocmask(SIG\_SETMASK,&orig\_mask,NULL);

/\*在critical region期间，如果有信号到达，解除阻塞后，如果在此处，信号被递送给进程，就会导致pause无法捕获到critical region期间的信号\*/

pause();

我们可以看到解除对特定信号的阻塞之后，调用pause之前，信号已经deliver给进程，这个信号已经错失了，pause无法等到这个信号，直到下一个信号deliver给进程pause都无法返回。这就违背了代码的本意：解除对信号的阻塞并且等待该信号第一次出现。

要避免这种情况，必须将解除信号阻塞和挂起进程等待信号两个动作封装成一个原子操作。这就是引入sigsuspend系统调用的原因。

### 6.8.2 sigsuspend函数

上一节中提到，pause之前传递信号是早期遇到的一个困境，并且没有好办法解决这个问题。本质上讲，讲解除信号的阻塞和挂起进程等待信号必须封装成原子操作，才能解决这个问题，sigsuspend解决了这个问题。

#include <signal.h>

int sigsuspend(const sigset\_t \*mask);

sigsuspend函数用mask指向的掩码来设置进程的阻塞掩码，并将进程挂起，直到进程捕捉到信号为止。一旦从信号处理函数中返回，sigsuspend函数会将进程的掩码恢复为调用之前的老的阻塞掩码值。

简单地说，sigsuspend相当于以不可中断的方式执行下面的操作：

sigprocmask(SIG\_SETMASK,&mask,&old\_mask);

pause();

sigprocmask(SIG\_SETMASK,&old\_mask,NULL);

有了sigsuspend函数，我们可以完成上一节pause完成不了的任务了：

static volatile sig\_atomic\_t sig\_received\_flag = 0;

sigset\_t mask\_all,mask\_most,mask\_old;

int signum = SIGUSR1;

sigfillset(&mask\_all);

sigfillset(&mask\_most);

sigdelset(&mask\_most,signum);

sigprocmask(SIG\_SETMASK,&mask\_all,&mask\_old);

/\*不要忘记先判断，因为sigprocmask阻塞所有信号之前，SIGUSR1可能被递送\*/

if(sig\_received\_flag == 0)

sigsuspend(&mask\_most);

sigprocmask(SIG\_SETMASK,&mask\_old,NULL);

假定等待特定信号SIGUSR1，首先讲所有的信号屏蔽掉，如果屏蔽信号之前，已经收到SIGUSR1，那么sig\_received\_flag会被设置为1，那么不需要调用sigsuspend，我们已经等到了我们要等的信号SIGUSR1。如果没收到，调用sigsuspend，阻塞掩码设为mask\_most，所有信号都屏蔽，只有SIGUSR1屏蔽。sigsuspend返回时，我们就可以确定，我们收到了信号SIGUSR1。阻塞掩码恢复成调用sigsuspend之前的mask\_all，然后显式将阻塞掩码恢复成默认的阻塞掩码mask\_old。

等一等，类似于上一节的代码，判断之后，pause之前，有信号递送，会导致信号错失，那么在上面的代码中，判断sig\_received\_flag==1之后，sigsuspend之前，是否会有SIGUSR1被递送给进程，导致错失信号一次？答案是否定的，因为我们已经setprocmask阻塞了所有信号，SIGUSR1没有机会递送给进程。

上面的代码完成了等待某特定信号的任务，但是上面的代码有副作用，就是等待期间，所有的其他信号都不能递送，原因是，sigsuspend的mask阻塞了所有SIGUSR1以外的信号，导致其他信号无法正常的递送。

static volatile sig\_atomic\_t sig\_received\_flag = 0;

sigset\_t mask\_blocked,mask\_old,mask\_unblocked;

int signum = SIGUSR1;

sigprocmask(SIGSETMASK,NULL,&mask\_blocked);

sigprocmask(SIGSETMASK,NULL,&mask\_unblocked);

sigaddset(&mask\_blocked,signum);

sigdelset(&mask\_unblocked,signum);

/\*将SIGUSR1添加到阻塞掩码中，确保下面判断sig\_received\_flag和sigsuspend之间不会收到SIGUSR1信号，导致SIGUSR1错失\*/

sigprocmask(SIG\_BLOCK,&mask\_blocked,&mask\_old);

/\*sigsuspend可能会有其他信号导致返回，所以需要判断sig\_received\_flag\*/

while(sig\_received\_flag == 0)

sigsuspend(&mask\_unblocked);

/\*将信号恢复成默认值\*/

sigprocmask(SIG\_SETMASK,&mask\_old,NULL);

上面的例子我们做到了等待特定信号SIGUSR1，期间如果有其他信号，不要影响其他信号的递送。至此等待特定信号算是圆满解决。

### 6.8.3 sigwaitinfo函数

sigsuspend函数可以实现等待特定信号的任务，但是看到上面示例过于繁复，有一定的心智负担，不够直接。sigwaitinfo函数可以同步的接收信号。

#include <signal.h>

int sigwaitinfo(const sigset\_t \*set, siginfo\_t \*info);

int sigtimedwait(const sigset\_t \*set, siginfo\_t \*info,

const struct timespec \*timeout);

sigwaitinfo函数会将当前进程挂起，知道set指向的信号集合中某个信号到达为止。如果调用调用sigwaitinfo函数时，set中已经有信号处于挂起状态，那么sigwaitinfo立即返回，传递来的信号从挂起队列中移除，并将返回信号的signo作为函数的返回值。info参数如果不是NULL，其中返回的信息和sigaction三参数信号处理函数中的第二参数siginfo\_t参数相同。

sigwaitinfo的本质是同步等待信号的到达，所以不需要编写信号处理函数。需要提示的一点是，纵然某信号遭到了阻塞，sigwaitinfo依然可以获取等待信号，所以调用sigwaitinfo之前，将set中信号阻塞是推荐的选择，因为如果不阻塞，调用sigwaitinfo之前或者两次sigwaitinfo之间有信号到达，只能按照默认的处理方式处理。

一个典型的调用sigwaitinfo的方式是如下的：

int sigusr1 = 0;

sigemptyset(&mask\_sigusr1);

sigaddset(&mask\_sigusr1,SIGUSR1);

sigprocmask(SIG\_SETMASK,&mask\_sigusr1,NULL);

while(1)

{

sig = sigwaitinfo(&mask\_sigusr1,&si);

sigusr1\_count++;

}

上面这个例子是统计收到SIGUSR1的次数。在sigwaitinfo之前，首先调用sigprocmask将等待集合的信号屏蔽。

sigtimedwait和sigwaitinfo用法是一样的，只不过timeout参数指定了最大等待的时长，如果调用超时但是没有等到任何信号，sigtimedwait返回-1，并且errno设为EAGAIN。

最后，SIGKILL和SIGSTOP不能等待，尝试等待SIGKILL和SIGSTOP会被忽略。原因和无法更改SIGKILL和SIGSTOP信号的信号处理函数一样，内核是终极boss，防止内核无法控制进程的情况出现。

## 6.9 通过文件描述符来获取信号

除了传统的信号处理函数和前面提到的通过sigwaitinfo同步地等待信号外，Linux提供了另外一种机制来接收信号：通过文件描述符来获取信号。这个机制和sigwaitinfo非常地类似，都属于同步等待信号的范畴，都需要首先调用sigprocmask将关注的信号屏蔽，防止被信号处理函数劫走。不同之处在于，文件描述符方法提供了文件系统的接口，可以通过select、poll和epoll来监控这些文件描述符。

下面看下接口：

#include <sys/signalfd.h>

int signalfd(int fd, const sigset\_t \*mask, int flags);

mask参数是信号集，表示关注的信号的集合。这些信号的集合应该在调用signalfd函数之前，首先调用sigprocmask函数，阻塞这些信号，防止被信号处理函数劫走。

首次创建时fd参数应该为-1，该函数会创建一个文件描述符，用于读取mask中到来的信号。如果fd不是-1，表示是修改操作，一般是修改mask的值。fd须是之前调用signalfd返回到值。

第三个参数 flags用来控制行为，目前支持的标志位有：

SFD\_CLOEXEC：和普通文件的O\_CLOEXEC一样，调用exec函数时，文件描述符会被关闭

SFD\_NONBLOCK：控制将来的读取操作，如果read时，并没有信号到来，立刻返回失败，设置errno为EAGAIN。

创建文件描述符后，可以使用read函数来读取到来的信号。提供缓冲区大小一般要足以放下一个signalfd\_siginfo结构体，该结构体一般包括如下成员变量

struct signalfd\_siginfo {

uint32\_t ssi\_signo;

int32\_t ssi\_errno;

int32\_t ssi\_code;

uint32\_t ssi\_pid;

uint32\_t ssi\_uid;

int32\_t ssi\_fd;

uint32\_t ssi\_tid;

uint32\_t ssi\_band;

uint32\_t ssi\_overrun;

uint32\_t ssi\_trapno;

int32\_t ssi\_status;

int32\_t ssi\_int;

uint64\_t ssi\_ptr;

uint64\_t ssi\_utime;

uint64\_t ssi\_stime;

uint64\_t ssi\_addr;

uint8\_t pad[X];

};

这个结构体和前面提到的siginfo\_t结构体几乎可以一一对应。含义和siginfo\_t中的成员一样的，不再赘述。

典型的使用signalfd来接收信号的方法如下（忽略了一些异常处理）：

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

sfd = signalfd(-1,&mask,NULL);

for(;;)

{

n = read(sfd,&fd\_siginfo,sizeof(struct signalfd\_siginfo));

if(n != sizeof(struct signalfd\_siginfo))

{

/\*error handle\*/

}

else

{

/\*process the signal\*/

}

}

用文件描述符和sigwaitinfo两种方法来处理信号是比较推荐的做法，因为传统信号处理函数的方法由于异步带来很多问题，其中用文件描述符的方法更加值得推荐，因为方法简单，可以和select、poll和epoll函数配合使用，非常灵活。

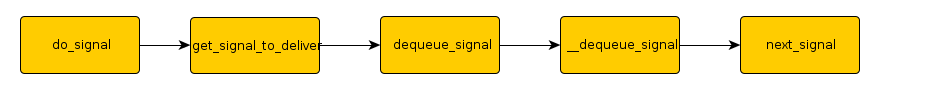
## 6.10信号递送的顺序

有一个非常有意思的话题，当处于挂起状态的信号有多个，信号递送的顺序是如何的呢？

信号实质上是软中断，中断有优先级，信号也有优先级。如果一个进程有多个未决信号，则对于同一个未决的实时信号，内核将按照发送的顺序来递送信号。如果存 在多个未决的实时信号，则值（或者说编号）越小的越先被递送。如果既存在不可靠信号，又存在可靠信号（实时信号），虽然POSIX对这一情况没有明确规 定，但Linux系统和大多数遵循POSIX标准的操作系统一样，将优先递送不可靠信号。

纵然是有限递送不可靠信号，不可靠信号不同信号之间的优先级如何呢？内核如何实现这些这些优先级的顺序呢？

内核选择信号deliver给进程的代码流程如下图所示：



int dequeue\_signal(struct task\_struct \*tsk,

sigset\_t \*mask, siginfo\_t \*info)

{

int signr;

/\* We only dequeue private signals from ourselves, we don't let

\* signalfd steal them

\*/

/\*线程私有的挂起信号队列优先\*/

signr = \_\_dequeue\_signal(&tsk->pending, mask, info);

if (!signr) {

signr = \_\_dequeue\_signal(&tsk->signal->shared\_pending,

mask, info);

...  
 }

前文我们讲过，线程的挂起信号队列有两个，一个是线程私有的pending，一个是线程组共享的signal->shared\_pending。如上面代码所示，选择信号deliver的特点是先从私有的pending中选择，然后从共享的shared\_pending中选择挂起信号递送给进程，当然选择的时候需要考虑线程的阻塞掩码。

int next\_signal(struct sigpending \*pending, sigset\_t \*mask)

{

unsigned long i, \*s, \*m, x;

int sig = 0;

s = pending->signal.sig;

m = mask->sig;

/\*

\* Handle the first word specially: it contains the

\* synchronous signals that need to be dequeued first.

\*/

x = \*s &~ \*m;

if (x) {

/\*优先选择同步信号，所谓同步信号集合就是SIGSEGV，SIGBUS等六种信号\*/

**if (x & SYNCHRONOUS\_MASK)**

x &= SYNCHRONOUS\_MASK;

/\*小信号值优先递送的算法\*/

sig = **ffz(~x)** + 1;

return sig;

}

switch (\_NSIG\_WORDS) {

default:

for (i = 1; i < \_NSIG\_WORDS; ++i) {

x = \*++s &~ \*++m;

if (!x)

continue;

sig = ffz(~x) + i\*\_NSIG\_BPW + 1;

break;

}

break;

case 2:

x = s[1] &~ m[1];

if (!x)

break;

sig = ffz(~x) + \_NSIG\_BPW + 1;

break;

case 1:

/\* Nothing to do \*/

break;

}

return sig;

}

#define SYNCHRONOUS\_MASK \

(sigmask(SIGSEGV) | sigmask(SIGBUS) | sigmask(SIGILL) | \

sigmask(SIGTRAP) | sigmask(SIGFPE) | sigmask(SIGSYS))

由于不同平台long的长度不同，所以算法略有不同，但是思想是一样的。

1. 出现在阻塞掩码的信号不能被选出。
2. 优先选择同步信号，所谓同步信号指的是一下6种信号

{SIGSEGV,SIGBUS,SIGILL,SIGTRAP,SIGFPE,SIGSYS}，

这六种信号都是与硬件相关的信号。

1. 如果没有上面六种信号，非实时信号优先，如果存在多种非实时信号，小信号值的信号优先。
2. 如果没有非实时信号，那么实时信号按照信号值递送，小信号值优先递送。

我们可以写测试程序来验证是否如此。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <string.h>

#include <errno.h>

static int sig\_cnt[NSIG];

static number= 0 ;

int sigorder[128]= {0};

#define MSG "#%d:receiver signal %d\n"

void handler(int signo)

{

/\*此处最好判断number的值，不要超出数组长度\*/

sigorder[number++] = signo;

}

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

{

int i = 0;

int k = 0;

sigset\_t blockall\_mask ;

sigset\_t pending\_mask ;

sigset\_t empty\_mask ;

struct sigaction sa ;

sigfillset(&blockall\_mask);

#ifdef USE\_SIGACTION

sa.sa\_handler = handler;

sa.sa\_mask = blockall\_mask ;

sa.sa\_flags = SA\_RESTART ;

#endif

printf("%s:PID is %d\n",argv[0],getpid());

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

{

if(i == SIGKILL || i == SIGSTOP)

continue;

#ifdef USE\_SIGACTION

if(sigaction(i,&sa, NULL)!=0)

#else

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

#endif

{

fprintf(stderr,"sigaction for signo(%d) failed (%s)\n",i,strerror(errno));

// return -1;

}

}

int sleep\_time = atoi(argv[1]);

if(sigprocmask(SIG\_SETMASK,&blockall\_mask,NULL) == -1)

{

fprintf(stderr,"setprocmask to block all signal failed(%s)\n",strerror(errno));

return -2;

}

printf("I will sleep %d second\n",sleep\_time);

sleep(sleep\_time);

sigemptyset(&empty\_mask);

if(sigprocmask(SIG\_SETMASK,&empty\_mask,NULL) == -1)

{

fprintf(stderr,"setprocmask to release all signal failed(%s)\n",strerror(errno));

return -3;

}

sleep(3)

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

{

if(sigorder[i] != 0)

{

printf("#%d: signo=%d\n",i,sigorder[i]);

}

}

return 0;

}

我们必须定义USE\_SIGACTION宏，因为我们在执行信号处理函数期间，我们要屏蔽到其他信号，否则无法验证信号的真实deliver顺序。

首先程序首先安装所有信号的信号处理函数（SIGKILL和SIGSTOP除外），然后阻塞所有信号，然后sleep 一段时间，在这段时间内，我们向进程发送各种信号，一旦解除阻塞，信号就会被递送到进程，进程就会执行信号处理函数。信号处理函数是精心定制的，按照递送的顺序，记录在静态数组中。我们按照顺序打印出信号的值，就获得了信号的递送顺序。

gcc -o sigaction\_delivery\_order -DUSE\_SIGACTION signal\_delivery\_order.c

我们可以看下测试程序的设计

#!/bin/bash

if [ $1 -eq 0 ]

then

./sigaction\_delivery\_order 30 &

else

./sigwait &

fi

signal\_pid=$!

sleep 2

kill -10 $signal\_pid

kill -3 $signal\_pid

kill -12 $signal\_pid

kill -11 $signal\_pid

kill -39 $signal\_pid

kill -2 $signal\_pid

kill -5 $signal\_pid

kill -4 $signal\_pid

kill -36 $signal\_pid

kill -24 $signal\_pid

kill -38 $signal\_pid

kill -37 $signal\_pid

kill -31 $signal\_pid

kill -8 $signal\_pid

kill -7 $signal\_pid

./tkill -p $signal\_pid -s 44

按照我们的设计，tkill是发向线程，信号会挂在私有的pending上，会优先递送，所以44号信号应该第一个被递送；其他的信号中4 = SIGILL，5=SIGTRAP，7=SIGBUS，8=SIGFPE，11=SIGSEGV，31=SIGSYS，这些都是属于同步信号集合，紧随44信号之后，按照从小到达，优先递送；2，3，10，12，24作为非实时信号，在随其后，被递送，最后是实时信号，按照从小到大，36，37，38，39，依次递送给进程。

我们看下是否按照这个顺序递送：

root@manu-hacks:~/code/c/self/signal# ./test\_order.sh 0

./sigaction\_delivery\_order:PID is 21897

sigaction for signo(32) failed (Invalid argument)

sigaction for signo(33) failed (Invalid argument)

I will sleep 30 second

root@manu-hacks:~/code/c/self/signal# #0: signo=44

#1: signo=4

#2: signo=5

#3: signo=7

#4: signo=8

#5: signo=11

#6: signo=31

#7: signo=2

#8: signo=3

#9: signo=10

#10: signo=12

#11: signo=24

#12: signo=36

#13: signo=37

#14: signo=38

#15: signo=39

和预想的一样，信号就是按照四个优先级递送到进程的。

## 6.11异步信号安全

当内核递送信号给进程时，进程正在执行的指令序列就会被中断，转而执行信号处理函数。待信号处理函数执行完毕返回（如果可以返回的话），则继续执行被中断的正常指令序列。这样的话，该进程就出现了两条控制流。

以errno为例，解释下可能带来的风险。每个线程都会有一个errno变量。如果主控制流刚刚设置了errno为某个值，这时候内核将某信号递送给进程，如果信号处理函数中，调用了其他系统调用发生错误，也可能会设置errno，这样当信号处理函数结束，跳回到主控制流时，errno已经被覆盖，再也无法获取正确的值。

因此，信号处理函数如果执行比较复杂的操作，一般建议首先要保存errno的值到某个临时变量，信号处理函数退出前，将errno恢复保存的值。

errno这个例子是比较简单，而且危害比较小的情况。设想malloc的情景：如果主程序流正在执行malloc函数中的某条指令，此时收到信号，在信号处理函数中也执行malloc，这种情况下程序很可能会陷入不可恢复的状态。首先的可能是死锁。malloc函数内部使用了锁，可以想象，如果主程序流已经获得了锁，信号处理函数中再次调用malloc申请锁，就会造成死锁：主程序流等待信号处理函数完成，信号处理函数等待主程序释放锁；其次如果信号处理函数中的malloc发现自己持有锁，直接操作，可能会破坏堆空间，使堆处于一种不一致的状态，最终使程序崩溃。

从上面的例子可以看出，信号处理函数中可以执行的函数要求很苛刻，能够在信号中安全调用的函数很少，这种函数称为满足异步信号安全的函数。man 7 signal可以获得异步信号安全函数的列表，在此就不罗列了（不罗列的一个原因是，并不鼓励在信号处理函数中执行真正有意义的工作）。

一般说来，不安全的函数大抵上可以分为以下几种情况

* 使用了静态变量，典型的是strtok，localtime等函数
* 使用了malloc或者free函数
* 标准I/O函数，如printf

编写信号处理函数是一件非常闹心的事情，因为一不小心就可能使用了不安全的函数，带来了隐患。一旦出现这种情况，排查起来非常的痛苦。所以当你编写了信号处理函数时，一定要保持如履薄冰的谨慎。也正是因为这个，有经验的程序员，不会在信号处理函数中执行很多的工作，一般仅仅是设置标志位，让主程序流去检查标志位，繁重的工作，交给主程序流去做。

另外一个解决的方法是通过sigwaitinfo函数和signalfd化异步为同步，减少出问题的可能。

## 总结

Linux的signal机制是一种原始的进程间通讯机制，传递的信息有限，很难传递复杂的消息，加上信号处理函数和进程处于两条执行逻辑流，会带来函数的重入问题，signal机制不适合作为进程间通信的主要手段。但是信号又不是完全无用，对于某些不频繁发生的异步事件，我们可以使用signal来通知进程。