第10章　信　　号

10.1　引言

信号是软件中断。很多比较重要的应用程序都需处理信号。信号提供了一种处理异步事件的方法：终端用户键入中断键，则会通过信号机构停止一个程序。

UNIX的早期版本，就已经有信号机构，但是这些系统，例如 V7所提供的信号模型并不可靠。信号可能被丢失，而且在执行临界区代码时，进程很难关闭所选择的信号。　4.3BSD和SVR3对信号模型都作了更改，增加了可靠信号机制。但是这两种更改之间并不兼容。幸运的是POSIX.1对可靠信号例程进行了标准化，这正是本章所说明的。

本章先对信号机制进行综述，并说明每种信号的一般用法。然后分析早期实现的问题。在分析存在的问题之后再说明解决这些问题的方法，这样有助于加深对改进机制的理解。本章也包含了很多并非100%正确的实例，这样做的目的是为了对其不足之处进行讨论。

10.2　信号的概念

首先，每个信号都有一个名字。这些名字都以三个字符 SIG开头。例如，SIGABRT是夭折信号，当进程调用abort函数时产生这种信号。SIGALRM是闹钟信号，当由alarm函数设置的时间已经超过后产生此信号。V7有15种不同的信号，SVR4和4.3+BSD均有31种不同的信号。

在头文件<signal.h>中，这些信号都被定义为正整数（信号编号）。没有一个信号其编号为0。在10.9节中将会看到kill函数，对信号编号0有特殊的应用。POSIX.1将此种信号编号值称为空信号。

很多条件可以产生一个信号。

• 当用户按某些终端键时，产生信号。在终端上按DELETE键通常产生中断信号（SIGINT）。这是停止一个已失去控制程序的方法。（第11章将说明此信号可被映射为终端上的任一字符。）

• 硬件异常产生信号：除数为0、无效的存储访问等等。这些条件通常由硬件检测到，并将其通知内核。然后内核为该条件发生时正在运行的进程产生适当的信号。例如，对执行一个无效存储访问的进程产生一个SIGSEGV。

• 进程用kill(2)函数可将信号发送给另一个进程或进程组。自然，有些限制：接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者必须是超级用户。

• 用户可用kill(1)命令将信号发送给其他进程。此程序是 kill函数的界面。常用此命令终止一个失控的后台进程。

• 当检测到某种软件条件已经发生，并将其通知有关进程时也产生信号。这里并不是指硬件产生条件（如被0除），而是软件条件。例如 SIGURG(在网络连接上传来非规定波特率的数据)、SIGPIPE(在管道的读进程已终止后一个进程写此管道)，以及SIGALRM(进程所设置的闹钟时间已经超时)。

信号是异步事件的经典实例。产生信号的事件对进程而言是随机出现的。进程不能只是测试一个变量(例如errno)来判别是否发生了一个信号，而是必须告诉内核“ 在此信号发生时，请

执行下列操作”。

可以要求系统在某个信号出现时按照下列三种方式中的一种进行操作。

(1) 忽略此信号。大多数信号都可使用这种方式进行处理，但有两种信号却决不能被忽略。它们是：SIGKILL和SIGSTOP。这两种信号不能被忽略的原因是：它们向超级用户提供一种使进程终止或停止的可靠方法。另外，如果忽略某些由硬件异常产生的信号（例如非法存储访问或除以0），则进程的行为是未定义的。

(2) 捕捉信号。为了做到这一点要通知内核在某种信号发生时，调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理。例如，若编写一个命令解释器，当用户用键盘产生中断信号时，很可能希望返回到程序的主循环，终止系统正在为该用户执行的命令。如果捕捉到SIGCHLD信号，则表示子进程已经终止，所以此信号的捕捉函数可以调用 waitpid 以取得该子进程的进程ID以及它的终止状态。又例如，如果进程创建了临时文件，那么可能要为SIGTERM信号编写一个信号捕捉函数以清除临时文件（kill命令传送的系统默认信号是终止信号）。

(3) 执行系统默认动作。表10-1给出了对每一种信号的系统默认动作。注意，对大多数信号的系统默认动作是终止该进程。

表10-1列出所有信号的名字，哪些系统支持此信号以及对于信号的系统默认动作。在POSIX.1列中，表示要求此种信号。job表示这是作业控制信号（仅当支持作业控制时，才要求此种信号）。

表10-1 UNIX信号

名 字	说 明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGABRT	异常终止(abort)	•	•	•	•	终止w/core
SIGALRM	超时(alarm)		•	•	•	终止
SIGBUS	硬件故障			•	•	终止w/core
SIGCHLD	子进程状态改变		作业	•	•	忽略
SIGCONT	使暂停进程继续		作业	•	•	继续/忽略
SIGEMT	硬件故障			•	•	终止w/core
SIGFPE	算术异常	•	•	•	•	终止w/core
SIGHUP	连接断开		•	•	•	终止
SIGILL	非法硬件指令	•	•	•	•	终止w/core
SIGINFO	键盘状态请求				•	忽略
SIGINT	终端中断符	•	•	•	•	终止
SIGIO	异步I/O			•	•	终止/忽略
SIGIOT	硬件故障			•	•	终止w/core
SIGKILL	终止		•	•	•	终止
SIGPIPE	写至无读进程的管道		•	•	•	终止
SIGPOLL	可轮询事件(poll)			•		终止
SIGPROF	梗概时间超时(setitimer)			•	•	终止
SIGPWR	电源失效/再起动			•		忽略
SIGQUIT	终端退出符		•	•	•	终止w/core
SIGSEGV	无效存储访问	•	•	•	•	终止w/core
SIGSTOP	停止		作业	•	•	暂停进程

（续）

名　字	说　明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGSYS	无效系统调用			•	•	终止w/core
SIGTERM	终止	•	•	•	•	终止
SIGTRAP	硬件故障			•	•	终止w/core
SIGTSTP	终端挂起符		作业	•	•	停止进程
SIGTTIN	后台从控制tty读		作业	•	•	停止进程
SIGTTOU	后台向控制tty写		作业	•	•	停止进程
SIGURG	紧急情况			•	•	忽略
SIGUSR1	用户定义信号		•	•	•	终止
SIGUSR2	用户定义信号		•	•	•	终止
SIGVTALRM	虚拟时间闹钟(setitimer)			•	•	终止
SIGWINCH	终端窗口大小改变			•	•	忽略
SIGXCPU	超过CPU限制(setrlimit)			•	•	终止w/core
SIGXFSZ	超过文件长度限制(setrlimit)			•	•	终止w/core

在系统默认动作列，“终止w/core”表示在进程当前工作目录的core文件中复制了该进程的存储图像（该文件名为core，由此可以看出这种功能很久之前就是UNIX功能的一部分）。大多数UNIX调试程序都使用core文件以检查进程在终止时的状态。在下列条件下不产生core文件：(a)进程是设置-用户-ID，而且当前用户并非程序文件的所有者，或者(b)进程是设置-组-ID，而且当前用户并非该程序文件的组所有者，或者(c)用户没有写当前工作目录的许可权，或者(d)文件太大(回忆7.11节中的RLIMIT_CORE)。core文件的许可权(假定该文件在此之前并不存在)通常是用户读／写，组读和其他读。

> core文件的产生不是POSIX.1所属部分，而是很多UNIX版本的实现特征。
>
> UNIX第6版没有检查条件(a)和(b)，并且其源代码中包含如下说明：“如果你正在找寻保护信号，那么当设置-用户-ID命令执行时，将可能产生大量的这种信号”。
>
> 4.3+BSD产生名为core.*prog*的文件，其中*prog*是被执行的程序名的前16个字符。它对core文件给予了某种标识，所以是一种改进特征。

表10-1“硬件故障”对应于实现定义的硬件故障。这些名字中有很多取自UNIX早先在PDP-11上的实现。请查看你所使用的系统的手册，以确切地确定这些信号对应于哪些错误类型。

下面比较详细地说明这些信号。

• SIGABRT　调用abort函数时（见10.17节）产生此信号。进程异常终止。

• SIGALRM　超过用alarm函数设置的时间时产生此信号。详细情况见10.10节。若由setitimer(2)函数设置的间隔时间已经过时，那么也产生此信号。

• SIGBUS　指示一个实现定义的硬件故障。

• SIGCHLD　在一个进程终止或停止时，SIGCHLD信号被送给其父进程。按系统默认，将忽略此信号。如果父进程希望了解其子进程的这种状态改变，则应捕捉此信号。信号捕捉函数中通常要调用wait函数以取得子进程ID和其终止状态。

系统V的早期版本有一个名为 SIGCLD(无H)的类似信号。这一信号具有非标准的语义，SVR2的手册页警告在新的程序中尽量不要使用这种信号。应用程序应当使用标准的 SIGCHLD信号。10.7节将讨论这两个信号。

• SIGCONT 此作业控制信号送给需要继续运行的处于停止状态的进程。如果接收到此信号的进程处于停止状态，则系统默认动作是使该进程继续运行，否则默认动作是忽略此信号。例如，vi编辑程序在捕捉到此信号后，重新绘制终端屏幕。关于进一步的情况见 10.20节。

• SIGEMT 指示一个实现定义的硬件故障。

> EMT这一名字来自PDP-11的emulator trap 指令。

• SIGFPE 此信号表示一个算术运算异常，例如除以0，浮点溢出等。

• SIGHUP 如果终端界面检测到一个连接断开，则将此信号送给与该终端相关的控制进程(对话期首进程)。见图9-11，此信号被送给session结构中s_leader字段所指向的进程。仅当终端的CLOCAL标志没有设置时，在上述条件下才产生此信号。(如果所连接的终端是本地的，才设置该终端的CLOCAL标志。它告诉终端驱动程序忽略所有调制解调器的状态行。第 11章将说明如何设置此标志。)注意，接到此信号的对话期首进程可能在后台，作为一个例子见图9-7。这区别于通常由终端产生的信号(中断、退出和挂起)，这些信号总是传递给前台进程组。

如果对话期前进程终止，则也产生此信号。在这种情况，此信号送给前台进程组中的每一个进程。

通常用此信号通知精灵进程（见第13章）以再读它们的配置文件。选用SIGHUP的理由是，因为一个精灵进程不会有一个控制终端，而且通常决不会接收到这种信号。

• SIGILL 此信号指示进程已执行一条非法硬件指令。

> 4.3BSD由abort函数产生此信号。SIGABRT现在被用于此。

• SIGINFO 这是一种4.3+BSD信号，当用户按状态键（一般采用Ctrl-T）时，终端驱动程序产生此信号并送至前台进程组中的每一个进程（见图 9-8)。此信号通常造成在终端上显示前台进程组中各进程的状态信息。

• SIGINT 当用户按中断键（一般采用DELETE或Ctrl-C）时，终端驱动程序产生此信号并送至前台进程组中的每一个进程(见图9-8)。当一个进程在运行时失控，特别是它正在屏幕上产生大量不需要的输出时，常用此信号终止它。

• SIGIO 此信号指示一个异步I/O事件。在12.6.2节中将对此进行讨论。

> 在表10-1中，对SIGIO的系统默认动作是终止或忽略。不幸的是，这依赖于系统。在SVR4中，SIGIO与SIGPOLL相同，其默认动作是终止此进程。在 4.3+BSD中(此信号起源于4.2BSD)，其默认动作是忽略。

• SIGIOT 这指示一个实现定义的硬件故障。

> IOT这个名字来自于PDP-11对于输入／输出TRAP(input/output TRAP)指令的缩写。系统V的早期版本，由abort函数产生此信号。SIGABRT现在被用于此。

• SIGKILL 这是两个不能被捕捉或忽略信号中的一个。它向系统管理员提供了一种可以

杀死任一进程的可靠方法。

• SIGPIPE　如果在读进程已终止时写管道，则产生此信号。14.2节将说明管道。当套接口的一端已经终止时，若进程写该套接口也产生此信号。

• SIGPOLL　这是一种SVR4信号，当在一个可轮询设备上发生一特定事件时产生此信号。12.5.2节将说明poll函数和此信号。它与4.3+BSD的SIGIO和SIGURG信号接近。

• SIGPROF　当setitimer(2)函数设置的梗概统计间隔时间已经超过时产生此信号。

• SIGPWR　这是一种SVR4信号，它依赖于系统。它主要用于具有不间断电源(UPS)的系统上。如果电源失效，则UPS起作用，而且通常软件会接到通知。在这种情况下，系统依靠蓄电池电源继续运行，所以无须作任何处理。但是如果蓄电池也将不能支持工作，则软件通常会再次接到通知，此时，它在15～30秒内使系统各部分都停止运行。此时应当传递SIGPWR信号。在大多数系统中使接到蓄电池电压过低的进程将信号SIGPWR发送给init进程，然后由init处理停机操作。很多系统V的init实现在inittab文件中提供了两个记录项用于此种目的；powerfail以及powerwait。

目前已能获得低价格的UPS系统，它用RS-232串行连接能够很容易地将蓄电池电压过低的条件通知系统，于是这种信号也就更加重要了。

• SIGQUIT　当用户在终端上按退出键（一般采用Ctrl-\）时，产生此信号，并送至前台进程组中的所有进程（见图9-8）。此信号不仅终止前台进程组（如SIGINT所做的那样），同时产生一个core文件。

• SIGSEGV　指示进程进行了一次无效的存储访问。

> 名字SEGV表示“段违例（segmentation violation）”。

• SIGSTOP　这是一个作业控制信号，它停止一个进程。它类似于交互停止信号(SIGTSTP)，但是SIGSTOP不能被捕捉或忽略。

• SIGSYS　指示一个无效的系统调用。由于某种未知原因，进程执行了一条系统调用指令，但其指示系统调用类型的参数却是无效的。

• SIGTERM　这是由kill(1)命令发送的系统默认终止信号。

• SIGTRAP　指示一个实现定义的硬件故障。

> 此信号名来自于PDP-11的TRAP指令。

• SIGTSTP　交互停止信号，当用户在终端上按挂起键[⊖]（一般采用Ctrl-Z）时，终端驱动程序产生此信号。

• SIGTTIN　当一个后台进程组进程试图读其控制终端时，终端驱动程序产生此信号。（见9.8节中对此问题的讨论。）在下列例外情形下不产生此信号，此时读操作返回出错，errno设置为EIO：(a)读进程忽略或阻塞此信号，或(b)读进程所属的进程组是孤儿进程组。

• SIGTTOU　当一个后台进程组进程试图写其控制终端时产生此信号。（见9.8节对此问题的讨论。）与上面所述的SIGTTIN信号不同，一个进程可以选择为允许后台进程写控制终端。第11章将讨论如何更改此选择项。

如果不允许后台进程写，则与SIGTTIN相似也有两种特殊情况：(a)写进程忽略或阻塞此信

⊖ 不幸的是，术语停止(stop)有不同的意义。在讨论作业控制和信号时我们需提及停止和继续作业。但是终端驱动程序一直用术语停止表示用Ctrl-S和Ctrl-Q字符停止和起动终端输出。因此，终端驱动程序将产生交互停止信号和字符称之为挂起字符而非停止字符。

号，或(b)写进程所属进程组是孤儿进程组。在这两种情况下不产生此信号，写操作返回出错，errno设置为EIO。

不论是否允许后台进程写，某些除写以外的下列终端操作也能产生此信号： tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow 以及tcsetpgrp。第11章将说明这些终端操作。

• SIGURG 此信号通知进程已经发生一个紧急情况。在网络连接上，接到非规定波特率的数据时，此信号可选择地产生。

• SIGUSR1 这是一个用户定义的信号，可用于应用程序。

• SIGUSR2 这是一个用户定义的信号，可用于应用程序。

• SIGVTALRM 当一个由setitimer(2)函数设置的虚拟间隔时间已经超过时产生此信号。

• SIGWINCH SVR4和4.3+BSD内核保持与每个终端或伪终端相关联的窗口的大小。一个进程可以用ioctl函数(见11.12节)得到或设置窗口的大小。如果一个进程用 ioctl的设置-窗口-大小命令更改了窗口大小，则内核将SIGWINCH信号送至前台进程组。

• SIGXCPU SVR4和4.3+BSD支持资源限制的概念（见 7.11节）。如果进程超过了其软CPU时间限制，则产生此信号。

• SIGXFSZ 如果进程超过了其软文件长度限制（见 7.11节），则SVR4和4.3+BSD产生此信号。

10.3 signal函数

UNIX信号机制最简单的界面是signal函数。

```
#include <signal.h>

void (*signal (int signo, void (func)(int))) (int);
```

返回：成功则为以前的信号处理配置，若出错则为 SIG_ERR

> signal函数由ANSI C定义。因为ANSI C不涉及多进程、进程组、终端I/O等，所以它对信号的定义非常含糊，以致于对 UNIX系统而言几乎毫无用处。确实，ANSI C对信号的说明只用了2页，而POSIX.1的说明则用了15页。
>
> SVR4也提供signal函数，该函数可提供老的SVR2不可靠信号语义（10.4节将说明这些老的语义）。提供此函数主要是为了向下兼容要求此老语义的应用程序，新应用程序不应使用它。
>
> 4.3+BSD也提供signal函数，但是它是用sigaction函数实现的（10.14节将说明sigaction函数），所以在4.3+BSD之下使用它提供新的可靠的信号语义。
>
> 在讨论sigaction函数时，提供了使用该函数的signal的一个实现。本书中的所有实例均使用程序10-12中给出的signal函数。

*signo*参数是表10-1中的信号名。*func*的值是：(a)常数SIG_IGN，或(b)常数SIG_DFL，或(c)当接到此信号后要调用的函数的地址。如果指定 SIG_IGN，则向内核表示忽略此信号。（记住有两个信号SIGKILL和SIGSTOP不能忽略。）如果指定SIG_DFL，则表示接到此信号后的动作是系统默认动作（见表10-1中的最后1列）。当指定函数地址时，我们称此为捕捉此信号。我们称此函数为信号处理程序（signal handler）或信号捕捉函数（signal-catching function）。

signal函数的原型说明此函数要求两个参数，返回一个函数指针，而该指针所指向的函数无返回值(void)。第一个参数*signo*是一个整型数，第二个参数是函数指针，它所指向的函数需要一个整型参数，无返回值。用一般语言来描述也就是要向信号处理程序传送一个整型参数，而它却无返回值。当调用signal设置信号处理程序时，第二个参数是指向该函数 (也就是信号处理程序)的指针。signal的返回值则是指向以前的信号处理程序的指针。

> 很多系统用附加的依赖于实现的参数来调用信号处理程序。 10.21节将说明可选择的SVR4和4.3+BSD参数。

本节开头所示的signal函数原型太复杂了，如果使用下面的 typedef〔Plauger 1992〕，则可使其简单一些。

```
typedef void    Sigfunc(int);
```

然后，可将signal函数原型写成：

```
Sigfunc *signal(int, Sigfunc *);
```

我们已将此typedef包括在ourhdr.h文件中（见附录B），并随本章中的函数一起使用。

如果查看系统的头文件<signal.h>, 则可能会找到下列形式的说明：

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

这些常数可用于表示“ 指向函数的指针，该函数要一个整型参数，而且无返回值 ”。signal的第二个参数及其返回值就可用它们表示。这些常数所使用的三个值不一定要是－ 1，0和1。它们必须是三个值而决不能是任一可说明函数的地址。大多数 UNIX系统使用上面所示的值。

实例

程序10-1显示了一个简单的信号处理程序，它捕捉两个用户定义的信号并打印信号编号。10.10节将说明pause函数，它使调用进程睡眠。

程序10-1　捕捉SIGUSR1和SIGUSR2的简单处理程序

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_usr(int);   /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");

    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
```

```
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        err_dump("received signal %d\n", signo);
    return;
}
```

我们使该程序在后台运行，并且用kill(1)命令将信号送给它。注意，在UNIX中，杀死(kill)这个术语是不恰当的。kill(1)命令和kill(2)函数只是将一个信号送给一个进程或进程组。该信号是否终止该进程则取决于该信号的类型，以及该进程是否安排了捕捉该信号。

```
$ a.out &                       在后台启动进程
[1]    4720                     作业控制shell打印作业号和进程ID
$ kill -USR1 4720               向该进程发送SIGUSR1
received SIGUSR1
$ kill -USR2 4720               向该进程发送SIGUSR2
received SIGUSR2
$ kill 4720                     向该进程发送SIGTERM
[1] + Terminated        a.out &
```

当向该进程发送SIGTERM信号后，该进程就终止，因为它不捕捉此信号，而对此信号的系统默认动作是终止。

10.3.1 程序起动

当执行一个程序时，所有信号的状态都是系统默认或忽略。通常所有信号都被设置为系统默认动作，除非调用exec的进程忽略该信号。比较特殊的是，exec函数将原先设置为要捕捉的信号都更改为默认动作，其他信号的状态则不变（一个进程原先要捕捉的信号，当其执行一个新程序后，就自然地不能再捕捉了，因为信号捕捉函数的地址很可能在所执行的新程序文件中已无意义）。

我们经常会碰到的一个具体例子是一个交互 shell如何处理对后台进程的中断和退出信号。对于一个非作业控制shell，当在后台执行一个进程时，例如：

```
cc main.c &
```

shell自动将后台进程中对中断和退出信号的处理方式设置为忽略。于是，当按中断键时就不会影响到后台进程。如果没有这样的处理，那么当按中断键时，它不但终止前台进程，也终止所有后台进程。

很多捕捉这两个信号的交互程序具有下列形式的代码：

```
int   sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

这样处理后，仅当SIGINT和SIGQUIT当前并不忽略，进程才捕捉它们。

从signal的这两个调用中也可以看到这种函数的限制：不改变信号的处理方式就不能确定信号的当前处理方式。我们将在本章的稍后部分说明使用 sigaction函数可以确定一个信号的处理方式，而无需改变它。

10.3.2 进程创建

当一个进程调用fork时，其子进程继承父进程的信号处理方式。因为子进程在开始时复制了父进程存储图像，所以信号捕捉函数的地址在子进程中是有意义的。

10.4 不可靠的信号

在早期的UNIX版本中（例如V7），信号是不可靠的。不可靠在这里指的是，信号可能会被丢失——一个信号发生了，但进程却决不会知道这一点。那时，进程对信号的控制能力也很低，它能捕捉信号或忽略它，但有些很需要的功能它却并不具备。例如，有时用户希望通知内核阻塞一信号——不要忽略该信号，在其发生时记住它，然后在进程作好了准备时再通知它。这种阻塞信号的能力当时并不具备。

> 4.2BSD对信号机构进行了更改，提供了被称之为可靠信号的机制。然后，SVR3也修改了信号机制，提供了另一套系统 V可靠信号机制。 POSIX.1选择了BSD模型作为其标准化的基础。

早期版本中的一个问题是在进程每次处理信号时，随即将信号动作复置为默认值 (在前面运行程序10-1时，我们通过只捕捉每种信号各一次避免了这一点)。以下是早期版本中关于如何处理中断信号的经典实例的代码：

```
    int     sig_int();          /* my signal handling function */

    ...
    signal(SIGINT, sig_int); /* establish handler */
    ...

sig_int()
{
    signal(SIGINT, sig_int);
                /* reestablish handler for next occurrence */

    ...                         /* process the signal ... */
}
```

由于早期的C语言版本不支持ANSI C的void数据类型，所以将信号处理程序说明为int类型。

这种代码段的一个问题是：在信号发生之后到信号处理程序中调用 signal函数之间有一个时间窗口。在此段时间中，可能发生另一次中断信号。第二个信号会造成执行默认动作，而对中断信号则是终止该进程。这种类型的程序段在大多数情况下会正常工作，使得我们认为它们正确，而实际上却并不是如此。

这些早期版本的另一个问题是：在进程不希望某种信号发生时，它不能关闭该信号。进程能做的就是忽略该信号。有时希望通知系统“ 阻止下列信号发生，如果它们确实产生了，请记住它们。”这种问题的一个经典实例是下列程序段，它捕捉一个信号，然后设置一个表示该信号已发生的标志：

```
int   sig_int_flag;           /* set nonzero when signal occurs */

main()
{
    int     sig_int();          /* my signal handling function */
    ...
```

```
    signal(SIGINT, sig_int); /* establish handler */
    ...
    while (sig_int_flag == 0)
        pause();             /* go to sleep, waiting for signal */
    ...
}

sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
    sig_int_flag = 1;        /* set flag for main loop to examine */
}
```

其中，进程调用pause函数使自己睡眠，直到捕捉到一个信号。当信号被捕捉到后，信号处理程序将标志sig_int_flag设置为非0。在信号处理程序返回之后，内核将该进程唤醒，它检测到该标志为非0，然后执行它所需做的。但是这里也有一个时间窗口，可能使操作错误。如果在测试sig_int_flag之后，调用paust之前发生信号，则此进程可能会一直睡眠（假定此信号不再次产生）。于是，这次发生的信号也就丢失了。还有另一个例子，某段代码并不正确，但是大多数时间却能正常工作。要查找并排除这种类型的问题很困难。

10.5 中断的系统调用

早期UNIX系统的一个特性是：如果在进程执行一个低速系统调用而阻塞期间捕捉到一个信号，则该系统调用就被中断不再继续执行。该系统调用返回出错，其 errno设置为EINTR。这样处理的理由是：因为一个信号发生了，进程捕捉到了它，这意味着已经发生了某种事情，所以是个好机会应当唤醒阻塞的系统调用。

> 在这里，我们必须区分系统调用和函数。当捕捉到某个信号时，被中断的是内核中执行的系统调用。

为了支持这种特性，将系统调用分成两类：低速系统调用和其他系统调用。低速系统调用是可能会使进程永远阻塞的一类系统调用，它们包括：

- 在读某些类型的文件时，如果数据并不存在则可能会使调用者永远阻塞（管道、终端设备以及网络设备）。
- 在写这些类型的文件时，如果不能立即接受这些数据，则也可能会使调用者永远阻塞。
- 打开文件，在某种条件发生之前也可能会使调用者阻塞（例如，打开终端设备，它要等待直到所连接的调制解调器回答了电话）。
- pause(按照定义，它使调用进程睡眠直至捕捉到一个信号)和wait。
- 某种ioctl操作。
- 某些进程间通信函数（见第14章）。

在这些低速系统调用中一个值得注意的例外是与磁盘 I/O有关的系统调用。虽然读、写一个磁盘文件可能暂时阻塞调用者(在磁盘驱动程序将请求排入队列, 然后在适当时间执行请求期间)，但是除非发生硬件错误，I/O操作总会很快返回，并使调用者不再处于阻塞状态。

可以用中断系统调用这种方法来处理的一种情况是：一个进程起动了读终端操作，而使用该终端设备的用户却离开该终端很长时间。在这种情况下进程可能处于阻塞状态几个小时甚至数天，除非系统停机，否则一直如此。

与被中断的系统调用相关的问题是必须用显式方法处理出错返回。典型的代码序列 (假定

进行一个读操作，它被中断，我们希望重新起动它)可能如下列样式：

```
again:
    if ( (n = read(fd, buff, BUFFSIZE)) < 0) {
       if (errno == EINTR)
           goto again;     /* just an interrupted system call */
       /* handle other errors */
    }
```

为了帮助应用程序使其不必处理被中断的系统调用， 4.2BSD引进了某些被中断的系统调用的自动再起动。自动再起动的系统调用包括： ioctl、read、readv、write、writev、wait和waitpid。正如前述，其中前五个函数只有对低速设备进行操作时才会被信号中断。而 wait和waitpid在捕捉到信号时总是被中断。某些应用程序并不希望这些函数被中断后再起动，因为这种自动再起动的处理方式也会带来问题，为此 4.3BSD允许进程在每个信号各别处理的基础上不使用此功能。

> POSIX.1允许实现再起动系统调用，但这并不是必需的。
>
> 系统V的默认工作方式是不再起动系统调用。但是 SVR4使用sigaction时（见10.14节），可以指定SA_RESTART选择项以再起动由该信号中断的系统调用。
>
> 在4.3+BSD中，系统调用的再起动依赖于调用了哪一个函数设置信号处理方式配置。早期的与4.3BSD兼容的sigvec函数使被该信号中断的系统调用自动再起动。但是，使用较新的与 POSIX.1兼容的sigaction则不使它们再起动。但如同在SVR4中一样，在sigaction中可以使用SA_RESTART选择项，使内核再起动由该信号中断的系统调用。

4.2BSD引进自动再起动功能的一个理由是：有时用户并不知道所使用的输入、输出设备是否是低速设备。如果我们编写的程序可以用交互方式运行，则它可能读、写终端低速设备。如果在程序中捕捉信号，而系统却不提供再起动功能，则对每次读、写系统调用就要进行是否出错返回的测试，如果是被中断的，则再进行读、写。

表10-2列出了几种实现所提供的信号功能及它们的语义。

表10-2 几种信号实现所提供的功能

函 数	系 统	信号处理程序仍被安装	阻塞信号的能力	被中断系统调用的自动再起动
signal,	V7, SVR2, SVR3, SVR4			决不
sigset,sighold,sigrelse sigignore,sigpause	SVR3, SVR4	•	•	决不
signal,sigvec,sigblock	4.2BSD	•	•	总是
sigsetmask,sigpause	4.3BSD, 4.3+BSD	•	•	默认
sigaction,sigprocmask sigpending,sigsuspend	POSIX.1	•	•	未说明
	SVR4	•	•	可选
	4.3+BSD	•	•	可选

应当了解，其他厂商提供的 UNIX系统可能会有不同于表 10-2中所示的处理情况。例如，SunOS 4.1.2中的sigaction其默认方式是再起动被中断的系统调用，这与 SVR4和4.3+BSD都不同。

程序10-12提供了我们自己的 signal函数版本，它试图重新起动被中断的系统调用（除SIGALRM信号外）。程序10-13则提供了另一个函数signal_intr，它不进行再起动。

在所有程序实例中，我们都有目的地显示了信号处理程序的返回 (如果它返回的话)，这种返回可能会中断一个系统调用。

12.5节说明select和poll函数时还将涉及被中断的系统调用。

10.6 可再入函数

进程捕捉到信号并继续执行时，它首先执行该信号处理程序中的指令。如果从信号处理程序返回（例如没有调用 exit或longjmp），则继续执行在捕捉到信号时进程正在执行的正常指令序列（这类似于硬件中断发生时所做的）。但在信号处理程序中，不能判断捕捉到信号时进程执行到何处。如果进程正在执行 malloc，在其堆中分配另外的存储空间，而此时由于捕捉到信号插入执行该信号处理程序，其中又调用 malloc，这时会发生什么 ?又例如若进程正在执行 getpwnam（见6.2节）这种将其结果存放在静态存储单元中的函数，而插入执行的信号处理程序中又调用这样的函数，这时又会发生什么呢 ? 在malloc例中子，可能会对进程造成破坏，因为malloc通常为它所分配的存储区保持一个连接表，而插入执行信号处理程序时，进程可能正在更改此连接表。在 getpwnam的例子中，正常返回给调用者的信息可能由返回至信号处理程序的信息覆盖。

POSIX.1说明了保证可再入的函数。表10-3列出了这些可再入函数。图中四个带 *号的函数并没有按POSIX.1说明为是可再入的，但SVR4 SVID 〔AT&T 1989〕则将它们列为是可再入的。

表10-3 信号处理程序中可以调用的可再入函数

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

没有列入表10-3中的大多数函数是不可再入的，其原因为：(a)已知它们使用静态数据结构，或(b)它们调用malloc或free，或(c)它们是标准I/O函数。标准I/O库的很多实现都以不可再入方式使用全局数据结构。

要了解在信号处理程序中即使调用列于表10-3中的函数，因为每个进程只有一个errno变量，所以我们可能修改了其原先的值。考虑一个信号处理程序，它恰好在 main刚设置errno之后被调用。如果该信号处理程序调用read，则它可能更改errno的值，从而取代了刚由main设置的值。因此，作为一个通用的规则，当在信号处理程序中调用表 10-3中列出的函数时，应当在其前保存，在其后恢复errno。(要了解经常被捕捉到的信号是SIGCHLD，其信号处理程序通常要调用一种wait函数，而各种wait函数都能改变errno。)

POSIX.1没有包括表10-3中的longjmp和siglongjmp（10.15节将说明siglongjmp函数）。这是因为在主例程以非再入方式正在更新一数据结构时可能产生信号。不是从信号处理程序返回而是调用siglongjmp，可能使该数据结构是部分更新的。如果应用程序将要做更新全局数据结构这样的事情，而同时规定要捕捉某些信号，而这些信号的处理程序又会引起执行 sigsetjmp，则在更新这种数据结构时要阻塞此信号。

实例

在程序10-2中，信号处理程序my_alarm调用不可再入函数getpwnam，而my_alarm每秒钟被调用一次。10.10节中将说明alarm函数。在程序10-2中用其每秒产生一次SIGALRM信号。

运行此程序时，其结果具有随意性。通常，在信号处理程序第一次返回时，该程序将由SIGSEGV信号终止。检查core文件，从中可以看到main函数已调用getpwnam，而且当信号处理程序调用此同一函数时，某些内部指针出了问题。偶然，此程序会运行若干秒，然后因产生SIGSEGV信号而终止。在捕捉到信号后，若 main函数仍正确运行，其返回值却有时错误，有时正确。有时在信号处理程序中调用 getpwnam会出错返回，其出错值为EBADF（无效文件描述符）。

从此实例中可以看出，若在信号处理程序中调用一个不可再入函数，则其结果是不可预见的。

程序10-2　在信号处理程序中调用不可再入函数

```
#include    <pwd.h>
#include    <signal.h>
#include    "ourhdr.h"

static void my_alarm(int);

int
main(void)
{
    struct passwd   *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);

    for ( ; ; ) {
        if ( (ptr = getpwnam("stevens")) == NULL)
            err_sys("getpwnam error");
        if (strcmp(ptr->pw_name, "stevens") != 0)
            printf("return value corrupted!, pw_name = %s\n",
                    ptr->pw_name);
    }
}
```

```
static void
my_alarm(int signo)
{
    struct passwd   *rootptr;

    printf("in signal handler\n");
    if ( (rootptr = getpwnam("root")) == NULL)
            err_sys("getpwnam(root) error");
    alarm(1);
    return;
}
```

10.7 SIGCLD语义

SIGCLD和SIGCHLD这两个信号经常易于混淆。SIGCLD是系统V的一个信号名，其语义与名为SIGCHLD的BSD信号不同。POSIX.1则采用BSD的SIGCHLD信号。

BSD的SIGCHLD信号的语义与其他信号的语义相类似。子进程状态改变后产生此信号，父进程需要调用一个wait函数以检测发生了什么。

由于历史原因，系统V处理SIGCLD信号的方式不同于其他信号。如果用signal或sigset（设置信号配置的早期的与SRV3兼容性函数）设置信号配置，则SVR4继续了这一具有问题色彩的传统（即兼容性限制）。对于SIGCLD早期的处理方式是：

(1) 如果进程特地指定对该信号的配置为SIG_IGN，则调用进程的子进程将不产生僵死进程。注意，这与其默认动作（SIG_DFL）忽略（见表10-1）不同。代之以，在子进程终止时，将其状态丢弃。如果调用进程最后调用一个wait函数，那么它将阻塞到所有子进程都终止，然后该wait会返回-1，其errno则设置为ECHILD。(此信号的默认配置是忽略，但这不会造成上述语义。代之以我们必须特地指定其配置为SIG_IGN。)

> POSIX.1并未说明在SIGCHLD被忽略时应产生的后果，所以这种行为是允许的。
>
> 4.3+BSD中，如SIGCHLD被忽略，则允许产生僵死子进程。如果要避免僵死子进程，则必须等待子进程。
>
> 在SVR4中，如果调用signal或sigset将SIGCHLD的配置设置为忽略，则不会产生僵死子进程。另外，使用SVR4版的sigaction，则可设置SA_NOCLDWAIT标志(见表10-5)以避免子进程僵死。

(2) 如果将SIGCLD的配置设置为捕捉，则内核立即检查是否有子进程准备好被等待，如果是这样，则调用SIGCLD处理程序。

第(2)项改变了为此信号编写处理程序的方法。

实例

10.4节曾提到进入信号处理程序后，首先要调用signal函数以再设置此信号处理程序。（在信号被复置为其默认值时，它可能被丢失，立即重新设置可以减少此窗口时间。）程序10-3显示了这一点。但此程序不能正常工作。如果在SVR2下编译并运行此程序，则其输出是一行行地不断重复“SIGCLD received”。最后进程用完其栈空间并异常终止。

此程序的问题是：在信号处理程序的开始处调用signal，按照上述第(2)项，内核检查是否有需要等待的子进程（因为我们正在处理一个SIGCLD，所以确实有这种子进程），

所以它产生另一个对信号处理程序的调用。信号处理程序调用 signal，整个过程再次重复。

为了解决这一问题，应当在调用 wait取到子进程的终止状态后再调用 signal。此时仅当其他子进程终止，内核才会再次产生此种信号。

> 如果为SIGCHLD建立了一个信号处理程序，又存在一个已终止但尚未等待的进程，则是否会产生信号？POSIX.1对此没有作说明。这样就允许前面所述的工作方式。但是，因为 POSIX.1在信号发生时并没有将信号配置复置为其默认值（假定正用POSIX.1的sigaction函数设置其配置），于是在SIGCHLD处理程序中也就不必再为该信号指定一个信号处理程序。

务必了解你所用的系统中SIGCHLD信号的语义。也应了解在某些系统中#define SIGCHLD为SIGCLD或反之。更改这种信号的名字使你可以编译为另一个系统编写的程序，但是如果该程序使用该信号的另一种语义，则这样的程序也不能工作。

程序10-3 不能正常工作的系统V SIGCLD处理程序

```
#include    <sys/types.h>
#include    <signal.h>
#include    <stdio.h>

static void sig_cld();

int
main()
{
    pid_t   pid;

    if (signal(SIGCLD, sig_cld) == -1)
        perror("signal error");

    if ( (pid = fork()) < 0)
        perror("fork error");
    else if (pid == 0) {        /* child */
        sleep(2);
        _exit(0);
    }
    pause();    /* parent */
    exit(0);
}

static void
sig_cld()
{
    pid_t   pid;
    int     status;

    printf("SIGCLD received\n");
    if (signal(SIGCLD, sig_cld) == -1)  /* reestablish handler */
        perror("signal error");

    if ( (pid = wait(&status)) < 0)     /* fetch child status */
        perror("wait error");
    printf("pid = %d\n", pid);
    return;     /* interrupts pause() */
}
```

10.8 可靠信号术语和语义

我们需要定义一些在讨论信号时会用到的术语。首先，当造成信号的事件发生时，为进程产生一个信号（或向一个进程发送一个信号）。事件可以是硬件异常（例如除以0）、软件条件（例如，闹钟时间超过）、终端产生的信号或调用kill函数。在产生了信号时，内核通常在进程表中设置某种形式的一个标志。当对信号做了这种动作时，我们说向一个进程递送了一个信号。在信号产生（generation）和递送（delivery）之间的时间间隔内，称信号未决（pending）。

进程可以选用“信号递送阻塞”。如果为进程产生了一个选择为阻塞的信号，而且对该信号的动作是系统默认动作或捕捉该信号，则为该进程将此信号保持为未决状态，直到该进程(a)对此信号解除了阻塞，或者(b)将对此信号的动作更改为忽略。当递送一个原来被阻塞的信号给进程时，而不是在产生该信号时，内核才决定对它的处理方式。于是进程在信号递送给它之前仍可改变对它的动作。进程调用sigpending函数（见10.13节）将指定的信号设置为阻塞和未决。

如果在进程解除对某个信号的阻塞之前，这种信号发生了多次，那么将如何呢?POSIX.1允许系统递送该信号一次或多次。如果递送该信号多次，则称这些信号排了队。但是大多数UNIX并不对信号排队。代之以，UNIX内核只递送这种信号一次。

> 系统V早期版本的手册页称SIGCLD信号是用排队方式处理的，但实际并非如此。代之以，内核按 10.7节中所述方式产生此信号。 AT&T〔1990e〕的sigaction(2)手册页称SA-SIGINFO标志（见表10-5）使信号可靠地排队，这也不正确。表面上此功能存在于内核中，但在SVR4中并不起作用。

如果有多个信号要递送给一个进程，那么将如何呢?POSIX.1并没有规定这些信号的递送顺序。但是POSIX.1建议：与进程当前状态有关的信号，例如SIGSEGV在其他信号之前递送。

每个进程都有一个信号屏蔽字，它规定了当前要阻塞递送到该进程的信号集。对于每种可能的信号，该屏蔽字中都有一位与之对应。对于某种信号，若其对应位已设置，则它当前是被阻塞的。进程可以调用 sigprocmask（在10.12节中说明）来检测和更改其当前信号屏蔽字。

信号数可能会超过一个整型数所包含的二进制位数，因此 POSIX.1定义了一个新数据类型sigset_t，它保持一个信号集。例如，信号屏蔽字就保存在这些信号集的一个中。 10.11节将说明对信号集进行操作的五个函数。

10.9 kill和raise函数

kill函数将信号发送给进程或进程组。raise函数则允许进程向自身发送信号。

> raise是由ANSI C而非POSIX.1定义的。因为ANSI C并不涉及多进程，所以它不能定义如kill这样要有一个进程ID作为其参数的函数。

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);
```

```
int raise(int signo);
```

两个函数返回：若成功则为0，若出错则为-1

kill的*pid*参数有四种不同的情况：

• *pid* > 0　将信号发送给进程ID为*pid*的进程。

• *pid* == 0　将信号发送给其进程组ID等于发送进程的进程组ID，而且发送进程有许可权向其发送信号的所有进程。

这里用的术语“所有进程”不包括实现定义的系统进程集。对于大多数UNIX系统，系统进程集包括：交换进程(pid 0)，init (pid 1)以及页精灵进程(pid 2)。

• *pid* < 0　将信号发送给其进程组ID等于*pid*绝对值，而且发送进程有许可权向其发送信号的所有进程。如上所述一样，“所有进程”并不包括系统进程集中的进程。

• *pid* == -1　POSIX.1未定义此种情况。

> SVR4和4.3+BSD用此广播信号（broadcast signal）。在广播信号时，并不把信号发送给上述系统进程集。4.3+BSD也不将广播信号发送给发送进程自身。若调用者是超级用户，则将信号发送给所有进程。如果调用者不是超级用户，则将信号发送给其实际用户ID或保存的设置-用户-ID等于调用者的实际或有效用户ID的所有进程。广播信号只能用于管理方面（例如一个超级用户进程使该系统停止运行）。

上面曾提及，进程将信号发送给其他进程需要许可权。超级用户可将信号发送给另一个进程。对于非超级用户，其基本规则是发送者的实际或有效用户ID必须等于接收者的实际或有效用户ID。如果实现支持_POSIX_SAVED_IDS（如SVR4所做的那样），则用保存的设置-用户-ID代替有效用户ID。

在对许可权进行测试时也有一个特例：如果被发送的信号是SIGCONT，则进程可将它发送给属于同一对话期的任一其他进程。

POSIX.1将信号编号0定义为空信号。如果signo参数是0，则kill仍执行正常的错误检查，但不发送信号。这常被用来确定一个特定进程是否仍旧存在。如果向一个并不存在的进程发送空信号，则kill返回-1，errno则被设置为ESRCH。但是，应当了解，UNIX系统在经过一定时间后会重新使用进程ID，所以一个现存的具有所给定进程ID的进程并不一定就是你所想要的进程。

如果调用kill为调用进程产生信号，而且此信号是不被阻塞的，那么在kill返回之前，*signo*或者某个其他未决的、非阻塞信号被传送至该进程。

10.10　alarm和pause函数

使用alarm函数可以设置一个时间值(闹钟时间)，在将来的某个时刻该时间值会被超过。当所设置的时间值被超过后，产生SIGALRM信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

返回：0或以前设置的闹钟时间的余留秒数

其中，参数*seconds*的值是秒数，经过了指定的*seconds*秒后会产生信号SIGALRM。要了解的是，经过了指定秒后，信号由内核产生，由于进程调度的延迟，进程得到控制能够处理该信号还需一段时间。

> 早期的UNIX版本曾提出警告，这种信号可能比预定值提前1秒发送。POSIX.1则不允许这样做。

每个进程只能有一个闹钟时间。如果在调用alarm时，以前已为该进程设置过闹钟时间，而且它还没有超时，则该闹钟时间的余留值作为本次alarm函数调用的值返回。以前登记的闹钟时间则被新值代换。

如果有以前登记的尚未超过的闹钟时间，而且*seconds*值是0，则取消以前的闹钟时间，其余留值仍作为函数的返回值。

虽然SIGALRM的默认动作是终止进程，但是大多数使用闹钟的进程捕捉此信号。如果此时进程要终止，则在终止之前它可以执行所需的清除操作。

pause函数使调用进程挂起直至捕捉到一个信号。

```
#include <unistd.h>

int pause(void);
```

返回：-1，errno设置为EINTR

只有执行了一个信号处理程序并从其返回时，pause才返回。在这种情况下，pause返回-1，errno设置为EINTR。

实例

使用alarm和pause，进程可使自己睡眠一段指定的时间。程序10-4中的sleep1函数提供这种功能。

程序10-4 sleep简化但并不完整的实现

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

static void
sig_alrm(int signo)
{
    return; /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);       /* start the timer */
    pause();            /* next caught signal wakes us up */
    return( alarm(0) ); /* turn off timer, return unslept time */
}
```

程序中的sleep1函数看起来与将在10.19节中说明的sleep函数类似，但这种简化实现有下列问题：

(1) 如果调用者已设置了闹钟，则它被sleep1函数中的第一次alarm调用擦去。

可用下列方法更正这一点：检查第一次调用 alarm的返回值，如其小于本次调用 alarm的参数值，则只应等到该前次设置的闹钟时间超时。如果前次设置闹钟时间的超时时刻后于本次设置值，则在sleep1函数返回之前，再次设置闹钟时间，使其在预定时间再发生超时。

(2) 该程序中修改了对SIGALRM的配置。如果编写了一个函数供其他函数调用，则在该函数被调用时先要保存原配置，在该函数返回前再恢复原配置。

更改这一点的方法是：保存signal函数的返回值，在返回前恢复设置原配置。

(3) 在调用alarm和pause之间有一个竞态条件。在一个繁忙的系统中，可能 alarm在调用pause之前超时，并调用了信号处理程序。如果发生了这种情况，则在调用 pause后，如果没有捕捉到其他信号，则调用者将永远被挂起。

sleep早期的实现与程序10-4类似，但更正了问题(1)和(2)。有两种方法可以更正问题(3)。第一种方法是使用 setjmp，下面立即说明这种方法。另一种方法是使用 sigprocmask和sigsuspend，10.19节将说明这种方法。

实例

SVR2中的sleep实现使用了setjmp和longjmp(见7.10节)以避免问题(3)中所说明的竞态条件。此函数的一个简化版本，称为sleep2，示于程序10-5中（为了缩短实例长度，程序中没有处理上面所说的问题(1)和(2)）。

程序10-5 sleep另一个不完善的实现

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

static jmp_buf  env_alrm;

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alrm) == 0) {
        alarm(nsecs);       /* start the timer */
        pause();            /* next caught signal wakes us up */
    }
    return( alarm(0) );     /* turn off timer, return unslept time */
}
```

在此函数中，程序10-4具有的竞态条件已被避免。即使pause从未执行，在发生SIGALRM时，sleep2函数也返回。

但是，sleep2函数中却有另一个难于察觉的问题，它涉及到与其他信号的相互作用。如果SIGALRM中断了某个其他信号处理程序，则调用 longjmp会提早终止该信号处理程序。程序10-6显示了这种情况。SIGINT处理程序中的for循环语句的执行时间在作者所用的系统上超过5秒钟，也就是大于sleep2的参数值，这正是我们想要的。整型变量j说明为volatile，这样就阻止

了优化编译程序除去循环语句。执行程序10-6得到：

```
$ a.out
^?                          键入中断字符
sig_int starting
sleep2 returned: 0
```

程序10-6 在一个捕捉其他信号的程序中调用sleep2

```
#include    <signal.h>
#include    "ourhdr.h"

unsigned int    sleep2(unsigned int);
static void     sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");

    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);

    exit(0);
}

static void
sig_int(int signo)
{
    int             i;
    volatile int    j;

    printf("\nsig_int starting\n");
    for (i = 0; i < 2000000; i++)
        j += i * i;
    printf("sig_int finished\n");
    return;
}
```

从中可见sleep2函数所引起的longjmp使另一个信号处理程序sig_int提早终止。如果将SVR2的sleep函数与其他信号处理程序一起使用，就可能碰到这种情况。见习题10.3。

sleep1和sleep2函数这两个实例的目的是告诉我们在涉及到信号时需要有精细而周到的考虑。下面几节将说明解决这些问题的方法，使我们能够可靠地，在不影响其他代码段的情况下处理信号。

实例

除了用来实现sleep函数外，alarm还常用于对可能阻塞的操作设置一个时间上限值。例如，程序中有一个读低速设备的会阻塞的操作（见10.5节），我们希望它超过一定时间量后就一定终止。程序10-7实现了这一点，它从标准输入读一行，然后将其写到标准输出上。

程序10-7 带时间限制调用read

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_alrm(int);
```

```
int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);

    exit(0);
}

static void
sig_alrm(int signo)
{
    return; /* nothing to do, just return to interrupt the read */
}
```

这种代码序列在很多UNIX应用程序中都能见到，但是这种程序有两个问题：

(1) 程序10-7有与程序10-4同样的问题：在第一次alarm调用和read调用之间有一个竞态条件。如果内核在这两个函数调用之间使进程不能占用处理机运行，而其时间长度又超过闹钟时间，则read可能永远阻塞。大多数这种类型的操作使用较长的闹钟时间，例如 1分钟或更长一点，使这种问题不会发生，但无论如何这是一个竞态条件。

(2) 如果系统调用是自动再起动的，则当从SIGALRM信号处理程序返回时， read并不被终止。在这种情形下，设置时间限制不会起作用。

在这里我们确实需要终止慢速系统调用。但是， POSIX.1并未提供一种可移植的方法实现这一点。

实例

让我们用longjmp再实现前面的实例(见程序10-8)。使用这种方法则无需担心一个慢速的系统调用是否被中断。

程序10-8 使用longjmp，带时间限制调用read

```
#include    <setjmp.h>
#include    <signal.h>
#include    "ourhdr.h"

static void     sig_alrm(int);
static jmp_buf  env_alrm;

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    if (setjmp(env_alrm) != 0)
        err_quit("read timeout");
```

```
    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);

    exit(0);
}

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

不管系统是否重新起动系统调用，该程序都会如所预期的那样工作。但是要知道，该程序仍旧有与程序10-5相同的与其他信号处理程序相互作用的问题。

如果要对I/O操作设置时间限制，则如上所示可以使用 longjmp，当然也要清楚它可能有与其他信号处理程序相互作用的问题。另一种选择是使用 select或poll函数，12.5.1节和12.5.2节将对它们进行说明。

10.11 信号集

我们需要有一个能表示多个信号——信号集（ signal set）的数据类型。将在 sigprocmask（下一节中说明）这样的函数中使用这种数据类型，以告诉内核不允许发生该信号集中的信号。如前所述，信号种类数目可能超过一个整型量所包含的位数，所以一般而言，不能用整型量中的一位代表一种信号。POSIX.1定义数据类型sigset_t以包含一个信号集，并且定义了下列五个处理信号集的函数。

```
#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);
```

四个函数返回：若成功则为0，若出错则为-1

```
int sigismember(const sigset_t *set, int signo);
```

返回：若真则为1，若假则为0

函数sigemptyset初始化由*set*指向的信号集，使排除其中所有信号。函数 sigfillset初始化由*set*指向的信号集，使其包括所有信号。所有应用程序在使用信号集前，要对该信号集调用 sigemptyset或sigfillset一次。这是因为C编译程序将不赋初值的外部和静态度量都初始化为 0，而这是否与给定系统上信号集的实现相对应并不清楚。

一旦已经初始化了一个信号集，以后就可在该信号集中增、删特定的信号。函数 sigaddset将一个信号添加到现存集中， sigdelset则从信号集中删除一个信号。对所有以信号集作为参数

的函数，都向其传送信号集地址。

实例

如果实现的信号数目少于一个整型量所包含的位数，则可用一位代表一个信号的方法实现信号集。例如，大多数4.3+BSD实现中有31种信号和32位整型。sigemptyset和sigfillset这两个函数可以在<signal.h>头文件中实现为宏：

```
#define sigemptyset(ptr)  ( *(ptr) = 0)
#define sigfillset(ptr)   ( *(ptr) = ~(sigset_t)0, 0 )
```

注意，除了设置对应信号集中各信号的位外，sigfillset必须返回0，所以使用逗号算符，它将逗号算符后的值作为表达式的值返回。

使用这种实现，sigaddset打开一位，sigdelset则关闭一位，sigismember测试一指定位。因为没有信号编号值为0，所以从信号编号中减1以得到要处理的位的位编号数。程序10-9可实现这些功能。

程序10-9 sigaddset、sigdelset和sigismember的实现

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

#define SIGBAD(signo)    ((signo) <= 0 || (signo) >= NSIG)
    /* <signal.h> usually defines NSIG to include signal number 0 */

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);       /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &= ~(1 << (signo - 1));    /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    return( (*set & (1 << (signo - 1))) != 0 );
}
```

也可将这三个函数在<signal.h>中实现为各一行的宏，但是POSIX.1要求检查信号编号参数的有效性，如果无效则设置errno。在宏中实现这一点比函数要难。

10.12 sigprocmask函数

10.8节曾说明一个进程的信号屏蔽字规定了当前阻塞而不能递送给该进程的信号集。调用函数sigprocmask可以检测或更改(或两者)进程的信号屏蔽字。

```
# include <signal.h>

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

返回：若成功则为0，若出错则为-1

首先，*oset*是非空指针，进程的当前信号屏蔽字通过 *oset*返回。其次，若*set*是一个非空指针，则参数*how*指示如何修改当前信号屏蔽字。表10-4说明了*how*可选用的值。SIG_BLOCK是或操作，而SIG_SETMASK则是赋值操作。

表10-4 用sigprocmask更改当前信号屏蔽字的方法

how	说 明
SIG_BLOCK	该进程新的信号屏蔽字是其当前信号屏蔽字和 *set*指向信号集的并集。*set*包含了我们希望阻塞的附加信号
SIG_UNBLOCK	该进程新的信号屏蔽字是其当前信号屏蔽字和 *set*所指向信号集的交集。*set*包含了我们希望解除阻塞的信号
SIG_SETMASK	该进程新的信号屏蔽是*set*指向的值

如果*set*是个空指针，则不改变该进程的信号屏蔽字，*how*的值也无意义。

如果在调用sigprocmask后有任何未决的、不再阻塞的信号，则在 sigprocmask返回前，至少将其中之一递送给该进程。

实例

程序10-10是一个函数，它打印调用进程的信号屏蔽字所阻塞信号的名称。从程序 10-14和10-15中调用此函数。为了节省空间，没有对表 10-1中列出的每一种信号测试该屏蔽字（见习题10.9）。

程序10-10 为进程打印信号屏蔽字

```
#include    <errno.h>
#include    <signal.h>
#include    "ourhdr.h"

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");
        /* remaining signals can go here */
    printf("\n");
    errno = errno_save;
}
```

10.13 sigpending函数

sigpending返回对于调用进程被阻塞不能递送和当前未决的信号集。该信号集通过 *set*参数返回。

```
#include <signal.h>

int sigpending(sigset_t *set);
```

返回：若成功则为0，若出错则为-1

实例

程序10-11使用了很多前面说明过的信号功能。进程阻塞了 SIGQUIT信号，保存了当前信号屏蔽字（以便以后恢复），然后睡眠5秒钟。在此期间所产生的退出信号都被阻塞，不递送至该进程，直到该信号不再被阻塞。在5秒睡眠结束后，检查是否有信号未决，然后将 SIGQUIT设置为不再阻塞。

注意，在设置SIGQUIT为阻塞时，我们保存了老的屏蔽字。为了解除对该信号的阻塞，用老的屏蔽字重新设置了进程信号屏蔽字（ SIG_SETMASK）。另一种方法是用SIG_UNBLOCK使阻塞的信号不再阻塞。但是，应当了解如果编写一个可能由其他人使用的函数，而且需要在函数中阻塞一个信号，则不能用SIG_UNBLOCK解除对此信号的阻塞，这是因为此函数的调用者在调用本函数之前可能也阻塞了此信号。在这种情况下必须使用 SIG_SETMASK将信号屏蔽字恢复为原先值。10.18节的system函数部分有这样的一个例子。

在睡眠期间如果产生了退出信号，那么此时该信号是未决的，但是不再受阻塞，所以在sigprocmask返回之前，它被递送到本进程。从程序的输出中可以看到这一点： SIGQUIT处理程序（sig_quit）中的printf语句先执行，然后再执行sigprocmask之后的printf语句。

程序10-11 信号设置和sigprocmask实例

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
                /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5);       /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
```

```
        printf("\nSIGQUIT pending\n");

                    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5);       /* SIGQUIT here will terminate with core file */

    exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");

    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
    return;
}
```

然后该进程再睡眠5秒钟。如果在此期间再产生退出信号，那么它就会使该进程终止，因为在上次捕捉到该信号时，已将其处理方式设置为默认动作。此时如果键入终端退出字符 Ctrl-\，则输出“ QUIT(coredump) ”信息，表示进程因接到SIGQUIT而终止，但是在core文件中保存了与进程有关的信息(该信息是由shell发现其子进程异常终止时打印的)。

```
$ a.out
^\                          产生信号一次(在5秒之内)
SIGQUIT pending             从sleep返回后
caught SIGQUIT              在信号处理程序中
SIGQUIT unblocked           从sigprocmask返回后
^\Quit(coredump)            再次产生信号
$ a.out
^\                          产生信号10次(在5秒之内)
SIGQUIT pending
caught SIGQUIT              只产生信号一次
SIGQUIT unblocked
^\Quit(coredump)            再产生信号
```

注意，在第二次运行该程序时，在进程睡眠期间使SIGQUIT信号产生了10次，但是解除了对该信号的阻塞后，只向进程传送一次SIGQUIT。从中可以看出在此系统上没有将信号进行排队。

10.14 sigaction函数

sigaction函数的功能是检查或修改(或两者)与指定信号相关联的处理动作。此函数取代了UNIX早期版本使用的signal函数。在本书末尾用sigaction函数实现了signal。

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
                         struct sigaction *oact);
```

返回：若成功则为0，若出错则为-1

其中，参数*signo*是要检测或修改具体动作的信号的编号数。若*act*指针非空，则要修改其动作。如果*oact*指针非空，则系统返回该信号的原先动作。此函数使用下列结构：

```
struct sigaction {
  void        (*sa_handler)(); /* addr of signal handler,
                                  or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask;             /* additional signals to block */
  int         sa_flags;         /* signal options, Table 10-5 */
};
```

当更改信号动作时，如果sa_handler指向一个信号捕捉函数(不是常数SIG_IGN或SIG_DFL)，则sa_mask字段说明了一个信号集，在调用信号捕捉函数之前，该信号集要加到进程的信号屏蔽字中。仅当从信号捕捉函数返回时再将进程的信号屏蔽字恢复为原先值。这样，在调用信号处理程序时就能阻塞某些信号。在信号处理程序被调用时，系统建立的新信号屏蔽字会自动包括正被递送的信号。因此保证了在处理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞到对前一个信号的处理结束为止。回忆 10.8节，若同一种信号多次发生，通常并不将它们排队，所以如果在某种信号被阻塞时，它发生了五次，那么对这种信号解除阻塞后，其信号处理函数通常只会被调用一次。

一旦对给定的信号设置了一个动作，那么在用 sigaction改变它之前，该设置就一直有效。这与早期的不可靠信号机制不同，而符合了POSIX.1在这方面的要求。

*act*结构的sa_flags字段包含了对信号进行处理的各个选择项。表 10-5详细列出了这些可选项的意义。

表10-5　信号处理的选择项标志(sa_flags)

可　选　项	POSIX.1	SVR4	4.3+BSD	说　明
SA_NOCLDSTOP	•	•	•	若*signo*是SIGCHLD，当一子进程停止时（作业控制），不产生此信号。当一子进程终止时，仍旧产生此信号（但请参阅下面说明的SVR4 SA_NOCLDWAIT可选项）
SA_RESTART		•	•	由此信号中断的系统调用自动再起动（参见 10.5节）
SA_ONSTACK		•	•	若用sigaltstack(2)已说明了一替换栈，则此信号递送给替换栈上的进程
SA_NOCLDWAIT		•		若*signo*是SIGCHLD，则当调用进程的子进程终止时，不创建僵死进程。若调用进程在后面调用 wait，则阻塞到它所有子进程都终止，此时返回 -1，errno设置为ECHILD（见10.7节）
SA_NODEFER		•		当捕捉到此信号时，在执行其信号捕捉函数时，系统不自动阻塞此信号。注意，此种类型的操作对应于早期的不可靠信号
SA_RESETHAND		•		对此信号的处理方式在此信号捕捉函数的入口处复置为SIG_DFL。注意，此种类型的信号对应于早期的不可靠信号
SA_SIGINFO		•		此选项对信号处理程序提供了附加信息。详细情况见10.21节

实例——signal函数

现在用sigaction实现signal函数。4.3+BSD也是这样做的(POSIX.1的原理阐述部分也说明这是POSIX所希望的)。SVR4则提供老的不可靠信号语义的 signal函数。除非为了向后兼容而使用老的语义，在SVR4之下，也应使用下面的 signal实现，或者直接调用 sigaction(在SVR4下，可以在调用sigaction时指定SA_RESETHAND和SA_NODEFER选择项以实现老的语义的signal函数)。本书中所有调用signal的实例均调用程序10-12中所实现的该函数。

程序10-12 用sigaction所实现的signal函数

```
/* Reliable version of signal(), using POSIX sigaction().  */

#include    <signal.h>
#include    "ourhdr.h"

Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;    /* SunOS */
#endif
    } else {
#ifdef  SA_RESTART
        act.sa_flags |= SA_RESTART;      /* SVR4, 4.3+BSD */
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

注意，必须用sigemptyset函数初始化*act*结构的成员。不能保证：

```
act.sa_mask = 0;
```

会做同样的事情。

我们对除SIGALRM以外的所有信号都企图设置SA_RESTART标志，于是被这些信号中断的系统调用都能再起动。不希望再起动由 SIGALRM信号中断的系统调用的原因是：我们希望对I/O操作可以设置时间限制。(请回忆与程序10-7有关的讨论。)

某些系统（如SunOS）定义了SA_INTERRUPT标志。这些系统的默认方式是重新起动被中断的系统调用，而指定此标志则使系统调用被中断后不再重起动。

实例——signal_intr函数

程序10-13是signal函数的另一种版本，它阻止被中断的系统调用再起动。

程序10-13 signal_intr函数

```
#include    <signal.h>
#include    "ourhdr.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
```

```
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef  SA_INTERRUPT    /* SunOS */
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

如果系统定义了SA_INTERRUPT，则在sa_flags中增加该标志，这样也就阻止了被中断的系统调用的再起动。

10.15 sigsetjmp和siglongjmp函数

7.10节说明了用于非局部转移的 setjmp和longjmp函数。在信号处理程序中经常调用longjmp函数以返回到程序的主循环中，而不是从该处理程序返回。确实， ANSI C标准说明一个信号处理程序可以返回或者调用 abort、exit或longjmp。程序10-5和10-8中已经有了这种情况。

调用longjmp时有一个问题。当捕捉到一个信号时，进入信号捕捉函数，此时当前信号被自动地加到进程的信号屏蔽字中。这阻止了后来产生的这种信号中断此信号处理程序。如果用longjmp跳出此信号处理程序，则对此进程的信号屏蔽字会发生什么呢？

> 在4.3+BSD下，setjmp和longjmp保存和恢复信号屏蔽字。但是， SVR4并不做这种操作。4.3+BSD提供函数_setjmp和_longjmp，它们也不保存和恢复信号屏蔽字。

为了允许两种形式并存， POSIX.1并没有说明setjmp和longjmp对信号屏蔽字的作用，而是定义了两个新函数sigsetjmp和siglongjmp。在信号处理程序中作非局部转移时应当使用这两个函数。

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);

                                返回：若直接调用则为0，若从siglongjmp调用返回则为非0

void siglongjmp(sigjmp_buf env, int val);
```

这两个函数和setjmp，longjmp之间的唯一区别是sigsetjmp增加了一个参数。如果*savemask*非0，则sigsetjmp在*env*中保存进程的当前信号屏蔽字。调用 siglongjmp时，如果带非0 *savemask*的sigsetjmp调用已经保存了*env*,则siglongjmp从其中恢复保存的信号屏蔽字。

实例

程序10-14显示了在信号处理程序被调用时，系统所设置的信号屏蔽字如何自动地包括刚被捕捉到的信号。它也例示了如何使用sigsetjmp和siglongjmp函数。

程序10-14 信号屏蔽、sigsetjmp和siglongjmp实例

```
#include    <signal.h>
#include    <setjmp.h>
#include    <time.h>
#include    "ourhdr.h"

static void                     sig_usr1(int), sig_alrm(int);
static sigjmp_buf               jmpbuf;
static volatile sig_atomic_t    canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    pr_mask("starting main: ");     /* Program 10.10 */

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("ending main: ");
        exit(0);
    }
    canjump = 1;    /* now sigsetjmp() is OK */

    for ( ; ; )
        pause();
}

static void
sig_usr1(int signo)
{
    time_t  starttime;

    if (canjump == 0)
        return;     /* unexpected signal, ignore */

    pr_mask("starting sig_usr1: ");

    alarm(3);               /* SIGALRM in 3 seconds */

    starttime = time(NULL);
    for ( ; ; )             /* busy wait for 5 seconds */
        if (time(NULL) > starttime + 5)
            break;

    pr_mask("finishing sig_usr1: ");

    canjump = 0;
    siglongjmp(jmpbuf, 1);  /* jump back to main, don't return */
}

static void
sig_alrm(int signo)
{
    pr_mask("in sig_alrm: ");
    return;
}
```

此程序例示了另一种技术，只要在信号处理程序中调用 siglongjmp就应使用这种技术。在调用sigsetjmp之后将变量canjump设置为非0。在信号处理程序中检测此变量，仅当它为非 0值时才调用siglongjmp。这提供了一种保护机制，使得若在jmpbuf(跳转缓存)尚未由sigsetjmp初始化时调用信号处理程序，则不执行其处理动作就返回（在本程序中， siglongjmp之后程序很快就结束，但是在较大的程序中，在siglongjmp之后，信号处理程序可能仍旧被设置）。在一般的

C代码中（不是信号处理程序），对于longjmp并不需要这种保护措施。但是，因为信号可能在任何时候发生，所以在信号处理程序中，需要这种保护措施。

在程序中使用了数据类型sig_atomic_t，它是由ANSI C定义的在写时不会被中断的变量类型。它意味着这种变量在具有虚存的系统上不会跨越页的边界，可以用一条机器指令对其进行存取。对于这种类型的变量总是包括 ANSI类型修饰符volatile，其原因是：该变量将由两个不同的控制线——main函数和异步执行的信号处理程序存取。

图10-1显示了此程序的执行时间顺序。可将图 10-1分成三部分：左面部分 (对应于main)，中间部分(sig_usr1)和右面部分(sig_alrm)。在进程执行左面部分时，信号屏蔽字是 0(没有信号是阻塞的)。而执行中间部分时，其信号屏蔽字是 SIGUSR1。执行右面部分时，信号屏蔽字是SIGUSR1｜SIGALRM。

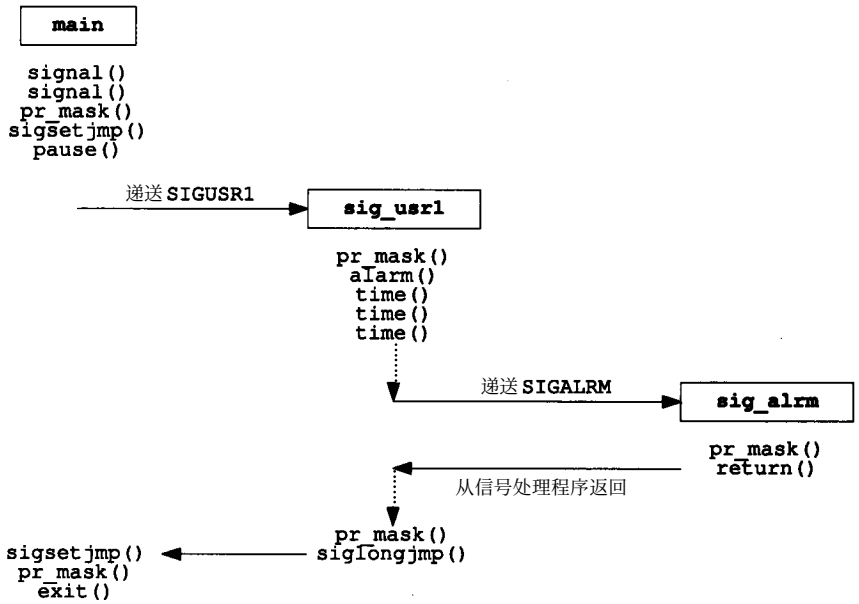

图10-1 处理两个信号的实例程序的时间顺序

执行程序10-14得到下面的输出：

```
$ a.out &                    在后台启动进程
starting main:
[1]   531                    作业控制shell打印其进程ID
$ kill -USR1 531             向该进程发送SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                             键入回车
[1] + Done          a.out &
```

该输出与我们所期望的相同：当调用一个信号处理程序时，被捕捉到的信号加到进程的当前信号屏蔽字。当从信号处理程序返回时，原来的屏蔽字被恢复。另外，siglongjmp恢复了由sigsetjmp所保存的信号屏蔽字。

如果将程序10-14中的sigsetjmp和siglongjmp分别代换成_setjmp和_longjmp，则在4.3+BSD下运行此程序，最后一行输出变成：

```
ending main: SIGUSR1
```

这意味着在调用 _setjmp之后执行main函数时，其SIGUSR1是阻塞的。这多半不是我们所希望的。

10.16 sigsuspend函数

上面已经说明，更改进程的信号屏蔽字可以阻塞或解除阻塞所选择的信号。使用这种技术可以保护不希望由信号中断的代码临界区。如果希望对一个信号解除阻塞，然后 pause以等待以前被阻塞的信号发生，则又将如何呢？假定信号是SIGINT，实现这一点的一种不正确的方法是：

```
sigset_t    newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
                /* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region of code */

                /* reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

pause();        /* wait for signal to occur */

/* continue processing */
```

如果在解除对SIGINT的阻塞和pause之间发生了SIGINT信号，则此信号被丢失。这是早期的不可靠信号机制的另一个问题。

为了纠正此问题，需要在一个原子操作中实现恢复信号屏蔽字，然后使进程睡眠，这种功能是由sigsuspend函数所提供的。

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

返回：-1，errno设置为EINTR

进程的信号屏蔽字设置为由 sigmask指向的值。在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程也被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则sigsuspend返回，并且该进程的信号屏蔽字设置为调用sigsuspend之前的值。

注意，此函数没有成功返回值。如果它返回到调用者，则总是返回－ 1，并且errno设置为EINTR(表示一个被中断的系统调用)。

实例

程序10-15显示了保护临界区，使其不被指定的信号中断的正确方法。

程序10-15 保护临界区不被信号中断

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

```
#include    "ourhdr.h"

static void sig_int(int);

int
main(void)
{
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);
                /* block SIGINT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

                /* critical region of code */
    pr_mask("in critical region: ");

                /* allow all signals and pause */
    if (sigsuspend(&zeromask) != -1)
        err_sys("sigsuspend error");
    pr_mask("after return from sigsuspend: ");

                /* reset signal mask which unblocks SIGINT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

                /* and continue processing ... */
    exit(0);
}

static void
sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
    return;
}
```

注意，当sigsuspend返回时，它将信号屏蔽字设置为调用它之前的值。在本例中， SIGINT信号将被阻塞。因此将信号屏蔽复置为早先保存的值(oldmask)。

运行程序10-15得到下面的输出：

```
$ a.out
in critical region: SIGINT
^?                                  键入自定义的中断字符
in sig_int: SIGINT
after return from sigsuspend: SIGINT
```

从中可见，在sigsuspend返回时，它将信号屏蔽字恢复为调用它之前的值。

实例

sigsuspend的另一种应用是等待一个信号处理程序设置一个全局变量。程序 10-16用于捕捉中断信号和退出信号，但是希望只有在捕捉到退出信号时再继续执行 main程序。此程序的样本输出是：

```
$ a.out
^?                      键入我们的中断字符
interrupt
^?                      再次键入我们的中断字符
interrupt
^?                      再一次
interrupt
^\ $                    用退出符终止
```

考虑到支持ANSI C的非POSIX系统与POSIX系统两者之间的可移植性，在一个信号处理程序中我们唯一应当做的是赋一个值给类型为 sig_atomic_t的变量。POSIX.1规定得更多一些，它说明了在一个信号处理程序中可以安全地调用的函数表（见表10-3），但是如果这样来编写代码，则它可能不会正确地在非 POSIX系统上运行。

程序10-16 用sigsuspend等待一个全局变量被设置

```
#include    <signal.h>
#include    "ourhdr.h"

volatile sig_atomic_t   quitflag;   /* set nonzero by signal handler */

int
main(void)
{
    void        sig_int(int);
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
                /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;
                /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    exit(0);
}

void
sig_int(int signo)  /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
```

```
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1;    /* set flag for main loop */
    return;
}
```

实例

可以用信号实现父子进程之间的同步，这是信号应用的另一个实例。程序 10-17实现了8.8节中提到的五个例程： TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT和WAIT_CHILD。其中使用了两个用户定义的信号： SIGUSR1由父进程发送给子进程， SIGUSR2由子进程发送给父进程。程序 14-3说明了使用管道的这五个函数的另一种实现。

程序10-17 父子进程可用来实现同步的例程

```
#include    <signal.h>
#include    "ourhdr.h"

static volatile sig_atomic_t    sigflag;
                                /* set nonzero by signal handler */
static sigset_t         newmask, oldmask, zeromask;

static void
sig_usr(int signo)  /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
    return;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);
        /* block SIGUSR1 and SIGUSR2, and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2);     /* tell parent we're done */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for parent */

    sigflag = 0;
```

```
            /* reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1);         /* tell child we're done */
}

void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for child */

    sigflag = 0;
            /* reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

如果在等待信号发生时希望去睡眠，则sigsuspend函数可以满足此种要求(正如在前面两个例子中所示)，但是如果在等待信号期间希望调用其他系统函数则将如何呢 ?不幸的是对此问题没有可靠的解决方法。

在程序17-13中我们遇到了这种情况。我们捕捉 SIGINT和SIGALRM这两种信号，在信号发生时，这两个信号处理程序都设置一个全局变量。用 signal-intr函数设置这两个信号处理程序，使得它们中断任一被阻塞的慢速系统调用。当阻塞在 select函数调用(见12.5.1节)，等待慢速设备的输入时很可能发生这两种信号（因为设置闹钟以阻止永远等待输入，所以这对SIGALRM是特别真实的）。我们能尽力做到的是：

```
if (intr_flag)      /* flag set by our SIGINT handler */
    handle_intr();
if (alrm_flag)      /* flag set by our SIGALRM handler */
    handle_alrm();
            /* signals occurring in here are lost */
while (select( ... ) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
            handle_intr();
    } else
        /* some other error */
}
```

在调用select之前测试各全局标志，如果 select返回一个中断的系统调用错误，则再次进行测试。如果在前两个if语句和后随的select调用之间捕捉到两个信号中的任意一个，则问题就发生了。正如代码中的注释所指出的，在此处发生的信号丢失了。调用相应的信号处理程序，它们设置了相应的全局变量，但是select决不会返回(除非某些数据已准备好可读)。

我们想要能够做的是下列步骤序列：

(1) 阻塞SIGINT和SIGALRM。

(2) 测试两个全局变量以判别是否发生了一个信号，如果已发生则处理此条件。

(3) 调用select(或任何其他系统调用，例如 read)并解除对这两个信号的阻塞，这两个操作要作为一个原子操作。

sigsuspend函数仅当第(3)步是一个pause操作时才帮助我们。

10.17 abort函数

前面已提及abort函数的功能是使程序异常终止。

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

此函数不返回

此函数将SIGABRT信号发送给调用进程。进程不应忽略此信号。

ANSI C要求若捕捉到此信号而且相应信号处理程序返回， abort仍不会返回到其调用者。如果捕捉到此信号，则信号处理程序不能返回的唯一方法是它调用 exit、_exit、longjmp或siglongjmp。(10.15节讨论了longjmp和siglongjmp之间的区别。）POSIX.1也说明abort覆盖了进程对此信号的阻塞和忽略。

让进程捕捉SIGABRT的意图是：在进程终止之前由其执行所需的清除操作。如果进程并不在信号处理程序中终止自己， POSIX.1说明当信号处理程序返回时， abort终止该进程。

ANSI C对此函数的规格说明将这一问题留由实现决定，而不管输出流是否刷新以及不管临时文件（见5.13节)是否删除。POSIX.1的要求则进了一步，它要求如果 abort调用终止进程，则它应该有对所有打开的标准I/O流调用fclose的效果。但是如果abort调用并不终止进程，则它对打开流也不应有影响。正如我们将在后面所看到的，这种要求很难实现。

> 系统V早期的版本中，abort函数产生SIGIOT信号。更进一步，进程忽略此信号，或者捕捉它并从信号处理程序返回都是可能的，在返回情况下， abort返回到它的调用者。
>
> 4.3BSD产生SIGILL信号。在此之前，该函数解除对此信号的阻塞，将其配置恢复为SIG_DFL (终止并构造core文件)。这阻止一个进程忽略或捕捉此信号。
>
> SVR4在产生此信号之前关闭所有 I/O流。在另一方面， 4.3+BSD则不做此操作。对于保护性的程序设计，如果希望刷新标准 I/O流，则在调用abort之前要做这种操作。在err_dump函数中实现了这一点（见附录B)。
>
> 因为大多数UNIX tmpfile(临时文件)的实现在创建该文件之后立即调用unlink，所以ANSI C关于临时文件的警告通常与我们无关。

实例

程序10-18按POSIX.1的说明实现了abort函数。对处理打开的标准I/O流的要求是难于实现的。首先查看是否执行了默认动作，并刷新了所有标准 I/O流。这并不等价于对所有打开的流调用fclose（因为只刷新，并不关闭它们)，但是当进程终止时，系统会关闭所有打开文件。如果进程捕捉此信号并返回，则刷新所有的流。(如果进程捕捉此信号，并且不返回，则不会触及标准I/O流。）没有处理的唯一条件是如果进程捕捉此信号，然后调用 _exit。在这种情况下，任何未刷新的存储器中的标准 I/O缓存都被丢弃。我们假定捕捉此信号，并特地调用 _exit的调用者并不想要刷新缓存。

回忆10.9节，如果调用kill使其为调用者产生信号，并且如果该信号是不被阻塞的（程序10-18保证做到了这一点)，则在kill返回前该信号就被传送给了该进程。这样就可确知如果对

kill的调用返回了，则该进程一定已捕捉到该信号，并且也从该信号处理程序返回。

程序10-18 abort的POSIX.1实现

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

void
abort(void)          /* POSIX-style abort() function */
{
    sigset_t            mask;
    struct sigaction    action;

        /* caller can't ignore SIGABRT, if so reset to default */
    sigaction(SIGABRT, NULL, &action);
    if (action.sa_handler == SIG_IGN) {
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);
    }

    if (action.sa_handler == SIG_DFL)
        fflush(NULL);           /* flush all open stdio streams */

        /* caller can't block SIGABRT; make sure it's unblocked */
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT);  /* mask has only SIGABRT turned off */
    sigprocmask(SIG_SETMASK, &mask, NULL);

    kill(getpid(), SIGABRT);    /* send the signal */

        /* if we're here, process caught SIGABRT and returned */
    fflush(NULL);               /* flush all open stdio streams */

    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);  /* reset disposition to default */
    sigprocmask(SIG_SETMASK, &mask, NULL);  /* just in case ... */

    kill(getpid(), SIGABRT);                /* and one more time */

    exit(1);    /* this should never be executed ... */
}
```

10.18 system函数

在8.12节已经有了一个system函数的实现，但是该版本并不做任何信号处理。 POSIX.2要求system忽略SIGINT和SIGQUIT，阻塞SIGCHLD。在给出一个正确地处理这些信号的一个版本之前，先说明为什么要考虑信号处理。

实例

程序10-19使用8.12节中的system版本，用其调用ed(1)编辑程序。(ed很久以来就是UNIX的组成部分。在这里使用它的原因是：它是一个交互式的捕捉中断和退出信号的程序。若从 shell调用ed，并键入中断字符，则它捕捉中断信号并打印问号。它也将对退出符的处理方式设置为忽略。)

程序10-19　用syetem调用ed编辑程序

```
#include    <sys/types.h>
#include    <signal.h>
#include    "ourhdr.h"

static void sig_int(int), sig_chld(int);

int
main(void)
{
    int     status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
        err_sys("signal(SIGCHLD) error");

    if ( (status = system("/bin/ed")) < 0)
        err_sys("system() error");
    exit(0);
}

static void
sig_int(int signo)
{
    printf("caught SIGINT\n");
    return;
}

static void
sig_chld(int signo)
{
    printf("caught SIGCHLD\n");
    return;
}
```

程序10-19用于捕捉SIGINT和SIGCHLD。若调用它则可得：

```
$ a.out
a                           将正文添加至编辑器缓存
Here is one line of text
and another
.                           行首的点停止添加方式
1, $p                       打印第1行至最后1行，以便观察缓存中的内容
Here is one line of text
and another
w temp.foo                  将缓存写至一文件
37                          编辑器称写了37个字节
q                           离开编辑器
caught SIGCHLD
```

当编辑程序终止时，对父进程(a.out进程)产生SIGCHLD信号。父进程捕捉它，执行其处理程序sig-chid，然后从其返回。但是若父进程正捕捉 SIGCHLD信号，那么在system函数执行时，父进程中该信号的递送应当阻塞。实际上，这就是 POSIX.2所说明的。否则，当system创建的子进程结束时，system的调用者可能错误地认为，它自己的一个子进程结束了。

如果再次执行该程序，在这次运行时将一个中断信号传送给编辑程序，则可得：

```
$ a.out
```

```
a                                   将正文添加至编辑器缓存
hello, world
.                                   行首的点停止添加方式
1,$p                                打印第1行至最后1行
hello, world
w etmp.foo                          将缓存写至一文件
13                                  编辑器称写了13个字节
^?                                  键入中断符
?                                   编辑器捕捉信号，打印问号
caught SIGINT                       父进程执行同一操作
q                                   离开编辑器
caught SIGCHLD
```

回忆9.6节可知，键入中断字符可使中断信号传送给前台进程组中的所有进程。图 10-2显示了编辑程序正在进行时的进程安排。

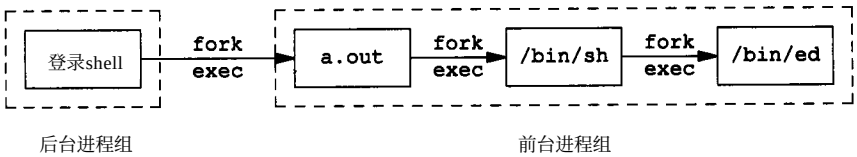

图10-2 程序10-19运行时的前、后台进程组

在这一实例中， SIGINT被送给三个前台进程。（shell进程忽略此信号。）从输出中可见，a.out进程和ed进程捕捉该信号。但是，当用 system运行另一个程序(例如ed)时，不应使父、子进程两者都捕捉终端产生的两个信号：中断和退出。这两个信号只应送给正在运行的程序：子进程。因为由system执行的命令可能是交互作用命令（如本例中的 ed），以及因为system的调用者在程序执行时放弃了控制，等待该执行程序的结束，所以 system的调用者就不应接收这两个终端产生的信号。这就是为什么POSIX.2规定system的调用者应当忽略这两个信号的原因。

实例

程序10-20是system函数的另一个实现，它进行了所要求的信号处理。

程序10-20 ystem函数的POSIX.2实现

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    <errno.h>
#include    <signal.h>
#include    <unistd.h>

int
system(const char *cmdstring)    /* with appropriate signal handling */
{
    pid_t               pid;
    int                 status;
    struct sigaction    ignore, saveintr, savequit;
    sigset_t            chldmask, savemask;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with Unix */

    ignore.sa_handler = SIG_IGN;    /* ignore SIGINT and SIGQUIT */
```

```
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);

    sigemptyset(&chldmask);          /* now block SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ( (pid = fork()) < 0) {
        status = -1;    /* probably out of processes */

    } else if (pid == 0) {          /* child */
            /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127);     /* exec error */

        } else {                        /* parent */
            while (waitpid(pid, &status, 0) < 0)
                if (errno != EINTR) {
                    status = -1; /* error other than EINTR from waitpid() */
                    break;
                }
        }

            /* restore previous signal actions & reset signal mask */
        if (sigaction(SIGINT, &saveintr, NULL) < 0)
            return(-1);
        if (sigaction(SIGQUIT, &savequit, NULL) < 0)
            return(-1);
        if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
            return(-1);

        return(status);
    }
```

很多较早的文献中使用下列程序段忽略中断和退出信号：

```
if ( (pid = fork()) < 0)
   err_sys("fork error");
else if (pid == 0) {    /* child */
   execl(…);
   _exit(127);
}
/* parent */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);
```

这段代码的问题是：在fork之后不能保证父进程还是子进程先运行。如果子进程先运行，父进程在一段时间后再运行，那么在父进程将中断信号的配置更改为忽略之前，就可能产生这种信号。由于这种原因，程序10-20在fork之前就改变对该信号的配置。

注意，子进程在调用execl之前要先恢复这两个信号的配置。这就允许在调用者配置的基础上，execl可将它们的配置更改为默认值。

system的返回值

system的返回值是shell的终止状态，它不总是执行命令字符串进程的终止状态。程序 8-13中有一些例子，其结果正是我们所期望的。如果执行一条如 date那样的简单命令，则其终止状态是0。执行shell命令exit 44，则得终止状态44。在信号方面又如何呢?

运行程序8-14，并向正在执行的命令发送一些信号。

```
$ tsys "sleep 30"
^?normal termination, exit status = 130      键入中断符
$ tsys "sleep 30"
^\sh: 946 Quit                               键入退出符
normal termination, exit status = 131
```

当用中断信号终止sleep时，pr_exit函数（见程序8-3）认为它正常终止。当用退出键杀死 sleep进程时，发生同样的事情。终止状态130、131又是怎样得到的呢?原来Bourne shell有一个在其文档中没有清楚说明的特性，其终止状态是 128加上它所执行的命令由一个信号终止时的该信号编号数。用交互方式使用shell可以看到这一点。

```
$ sh                                         确保运行Bourne shell
$ sh -c "sleep 30"
^?                                           键入中断符
$ echo $?                                    打印最后一条命令的终止状态
130
$ sh -c "sleep 30"
^\sh:962 Quit-core dumped                    键入退出符
$ echo $?                                    打印最后一条命令的终止状态
131
$ exit                                       离开Bourne shell
```

在所使用的系统中，SIGINT的值为2，SIGQUIT的值为3，于是给出shell终止状态130、131。

再说明几个类似的例子，这一次将一个信号直接送给shell，然后观察system返回什么。

```
$ tsys "sleep 30" &                          这一次在后台启动它
[1]     980
$ ps                                         查看进程ID
  PID TT STAT TIME COMMAND
  980 p3 S    0:00 tsys sleep 30
  981 p3 S    0:00 sh -c sleep 30
  982 p3 S    0:00 sleep 30
  985 p3 R    0:00 ps
$ kill -KILL 981                             杀死shell自身
abnormal termination, signal number = 9
[1] +  Done       tsys "sleep 30" &
```

从中可见仅当shell本身异常终止时，system的返回值才报告一个异常终止。

在编写使用system函数的程序时，一定要正确地解释返回值。如果直接调用 fork、exec和wait，则终止状态与调用system是不同的。

10.19 sleep函数

在本书的很多例子中都已使用了 sheep函数，在程序 10-4和10-5中有两个sleep的很完善的实现。

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

返回：0或未睡的秒数

此函数使调用进程被挂起直到：

(1) 已经过了*seconds*所指定的墙上时钟时间，或者

(2) 该进程捕捉到一个信号并从信号处理程序返回。

如同alarm信号一样，由于某些系统活动，实际返回时间比所要求的会迟一些。

在第(1)种情形，返回值是0。当由于捕捉到某个信号sleep提早返回时（第(2)种情形），返回值是未睡足的秒数(所要求的时间减实际睡眠时间)。

sleep可以用alarm函数（见10.10节）实现，但这并不是必需的。如果使用 alarm，则这两个函数之间可以有交互作用。 POSIX.1标准对这些交互作用并未作任何说明。例如，若先调用alarm(10)，过了3秒后又调用sleep(5)，那么将如何呢？sleep将在5秒后返回（假定在这段时间内没有捕捉到另一个信号)，但是否在2秒后又产生另一个 SIGALRM信号呢?这种细节依赖于实现。

> SVR4用alarm实现sleep。sleep(3)手册页中说明以前安排的闹钟仍被正常处理。例如，在前面的例子中，在 sleep返回之前，它安排在 2秒后再次到达闹钟时间。在这种情况下，sleep返回0。（很明显，sleep必须保存SIGALRM信号处理程序的地址，在返回前重新设置它。）另外，如果先做一次alarm(6), 3秒钟之后又做一次sleep(5)，则在3秒后sleep返回，而不是5秒钟。此时，sleep的返回值则是未睡足的时间2秒。
>
> 4.3+BSD则使用另一种技术：由setitimer(2)提供间隔计时器。该计时器独立于alarm函数，但在以前设置的间隔计时器和 sleep之间仍能有相互作用。另外，尽管闹钟计时器 (alarm)和间隔计时器 (setitimer)是分开的，但是不幸它们使用同一SIGALRM信号。因为sleep暂时将该信号的处理程序改变为它自己的函数，所以在alarm和sleep之间仍可能有不希望的相互作用。
>
> 如果混合调用sleep和其他与时间有关的函数，它们之间有相互作用，则你应当清楚地了解你所使用的系统是如何实现 sleep的。
>
> 以前伯克利类的sleep实现不提供任何有用的返回信息。这在 4.3+BSD中已经解决。

实例

程序10-21是一个POSIX.1 sleep函数的实现。此函数是程序 10-4的修改版，它可靠地处理信号，避免了早期实现中的竞态条件，但是仍未处理与以前设置的闹钟的相互作用（正如前面提到的，POSIX.1并未对这些交互作用进行定义)。

程序10-21 sleep的可靠实现

```
#include    <signal.h>
#include    <stddef.h>
#include    "ourhdr.h"

static void
sig_alrm(void)
{
    return; /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int
sleep(unsigned int nsecs)
{
    struct sigaction    newact, oldact;
    sigset_t            newmask, oldmask, suspmask;
    unsigned int        unslept;

    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);
                /* set our handler, save previous information */

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
                /* block SIGALRM and save current signal mask */
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(nsecs);

    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);  /* make sure SIGALRM isn't blocked */

    sigsuspend(&suspmask);          /* wait for any signal to be caught */

    /* some signal has been caught, SIGALRM is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);  /* reset previous action */

                /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);

    return(unslept);
}
```

与程序10-4相比，为了可靠地实现sleep，程序10-21的代码比较长。程序中没有使用任何形式的非局部转移（如程序10-5为了避免在alarm和pause之间的竞态条件所做的那样），所以对处理SIGALRM信号期间可能执行的其他信号处理程序没有影响。

10.20 作业控制信号

在表10-1中有六个POSIX.1认为是与作业控制有关的信号。

- SIGCHLD 子进程已停止或终止。
- SIGCONT 如果进程已停止，则使其继续运行。
- SIGSTOP 停止信号（不能被捕捉或忽略）。
- SIGTSTP 交互停止信号。
- SIGTTIN 后台进程组的成员读控制终端。

• SIGTTOU 后台进程组的成员写控制终端。

> 虽然仅当系统支持作业控制时，POSIX.1才要求它支持SIGCHLD，但是几乎所有UNIX版本都支持这种信号。我们已经说明了在子进程终止时这种信号是如何产生的。

大多数应用程序并不处理这些信号——交互式shell通常做处理这些信号的所有工作。当键入挂起字符（通常是Ctrl-Z）时，SIGTSTP被送至后台进程组的所有进程。当通知shell在前台或后台恢复一个作业时，shell向作业中的所有进程发送SIGCONT信号。与之类似的有，如果向一个进程递送了SIGTTIN或SIGTTOU信号，则根据系统默认，此进程停止，作业控制shell了解到这一点后就通知我们。

一个例外是管理终端的进程——例如，vi(1)编辑程序。当用户要挂起它时，它需要能了解到这一点，这样就能将终端状态恢复到vi起动时的情况。另外，当在前台恢复它时，它需要将终端状态设置回所希望的状态，并需要重新绘制终端屏幕。可以在下面的例子中观察到类似vi这样的程序是如何处理这种情况的。

在作业控制信号间有某种相互作用。当对一个进程产生四种停止信号（SIGTSTP，SIGSTOP，SIGTTIN或SIGTTOU）中的任意一种时，对该进程的任一未决的SIGCONT信号就被丢弃。与之类似的是，当对一个进程产生SIGCONT信号时，对同一进程的任一未决的停止信号被丢弃。

注意，如果进程是停止的，SIGCONT的默认动作是继续一个进程，否则忽略此信号。通常，对该信号无需做任何事情。当对一个停止的进程产生一个SIGCONT信号时，该进程就继续，即使该信号是被阻塞或忽略的也是如此。

实例

程序10-22例示了当一个程序处理作业控制时所使用的通常的代码序列。该程序只是将其标准输入复制到其标准输出，但是在信号处理程序中以注释形式给出了管理屏幕的程序所执行的典型操作。当程序10-22起动时，仅当SIGTSTP信号的配置是SIG_DFL，它才安排捕捉该信号。其理由是：当此程序由不支持作业控制的shell(例如/bin/sh)所起动时，此信号的配置应当设置为SIG_IGN。实际上，shell并不显式地忽略此信号，而是init将这三个作业控制信号SIGTSTP、SIGTTIN和SIGTTOU设置为SIG_IGN。然后，这种配置由所有登录shell继承。只有作业控制shell才应将这三个信号重新设置为SIG_DFL。

当键入挂起字符时，进程接到SIGTSTP信号，然后该信号处理被调用。在此点上，应当进行与终端有关的处理：将光标移到左下角，恢复终端工作方式，等等。在将SIGTSTP重新设置为默认值(停止该进程)，并且解除了对此信号的阻塞之后，进程向自己发送同一信号SIGTSTP。因为正在处理SIGTSTP信号，而在捕捉到该信号期间系统自动地阻塞它，所以应当解除对此信号的阻塞。仅当某个进程（通常是正响应一个交互式fg命令的作业控制shell）向该进程发送一个SIGCONT信号时，该进程才继续。我们不捕捉SIGCONT信号。该信号的默认配置是继续停止的进程，当此发生时，此程序如同从kill函数返回一样继续运行。当此程序继续运行时，将SIGTSTP信号再设置为捕捉，并且做我们所希望做的终端处理（例如重新绘制屏幕）。

第18章将介绍处理特定的作业控制挂起字符的另一种方法，其中并不使用信号，而是由程序自身识别该特定字符。

程序10-22 如何处理SIGTSTP

```
#include    <signal.h>
#include    "ourhdr.h"

#define BUFFSIZE    1024

static void sig_tstp(int);

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    /* only catch SIGTSTP if we're running with a job-control shell */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
    if (n < 0)
        err_sys("read error");

    exit(0);
}

static void
sig_tstp(int signo) /* signal handler for SIGTSTP */
{
    sigset_t    mask;

    /* ... move cursor to lower left corner, reset tty mode ... */

    /* unblock SIGTSTP, since it's blocked while we're handling it */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL);   /* reset disposition to default */

    kill(getpid(), SIGTSTP);    /* and send the signal to ourself */

        /* we won't return from the kill until we're continued */

    signal(SIGTSTP, sig_tstp);  /* reestablish signal handler */

    /* ... reset tty mode, redraw screen ... */
    return;
}
```

10.21 其他特征

本节介绍某些依赖于实现的信号的其他特征。

10.21.1 信号名字

某些系统提供数组

```
extern char *sys_siglist[];
```

数组下标是信号编号，数组中的元素是指向一个信号字符串名字的指针。

这些系统通常也提供函数psignal。

```
#include <signal.h>
void psignal(int signo, const char.* msg);
```

字符串*msg* (通常是程序名)输出到标准出错，后面跟着一个冒号和一个空格，再跟着对该信号的说明，最后是一个新行符。

SVR4和4.3+BSD都提供sys_siglist和psignal函数。

10.21.2 SVR4信号处理程序的附加参数

当调用sigaction对一个信号设置配置时，可以指定sa_flags值SA_SIGINFO (见表10-5)。这使得两个附加参数被传给信号处理程序。整型的信号编号总是作为第一个参数传送。第二个参数是一个空指针，或者是一个指向 siginfo结构的指针。(第三个参数提供进程内不同控制线程的有关信息，在此不对它进行讨论。)

```
struct siginfo {
  int    si_signo;  /* signal number */
  int    si_errno;  /* if nonzero, errno value from <errno.h> */
  int    si_code;   /* additional info (depends on signal) */
  pid_t  si_pid;    /* sending process ID */
  uid_t  si_uid;    /* sending process real user ID */
  /* other fields also */
};
```

对于由硬件产生的信号，例如 SIGFPE，si_code值给出附加的信息：FPE_INTDIV表示整数除以0，FPE_FLTDIV表示浮点数除以0等等。如果si_code小于或等于0，则表示信号是由调用kill(2)的用户进程产生的。在此情况下， si_pid和si_uid给出了发送此信号的进程的有关信息。依赖于正被捕捉的信号，还有一些信息可用，见SVR4 siginfo(5)手册页。

10.21.3 4.3+BSD信号处理程序的附加参数

4.3+BSD总是用三个参数调用信号处理程序：

```
handler(int signo, int code, struct sigcontext *scp);
```

参数*signo*是信号编号， *code*给出某些信号的附加信息。例如，对于 SIGFPE的*code*值FPE_INTDIV_TRAP表示整数除以0。第三个参数*scp*与硬件有关。

10.22 小结

信号用于很多复杂的应用程序中。理解进行信号处理的原因和方式对于高级 UNIX程序设计极其重要。本章对UNIX信号进行了详细而且比较深入的介绍。首先说明了早期的信号实施的问题以及它们是如何显现出来的。然后介绍了POSIX.1的可靠信号概念以及所有相关的函数。在此基础上接着提供了abort、system和sleep函数的POSIX.1实现。最后以观察分析作业控制信号结束。

习题

10.1 删除程序10-1中的for (; ;) 语句，结果会怎样？为什么？

10.2 实现raise函数。

10.3 画出运行程序10-6时的堆栈情况。

10.4 程序10-8中利用setjmp和longjmp设置I/O操作的超时，下面的代码也常用于超时：

```
signal(SIGALRM, sig_alrm);
alarm(60) ;
if (setjmp (env_alrm) != 0) {
     /* handle time out */
     ...
     }
...
```

这段代码有什么错误？

10.5 仅使用一个计时器(alarm或较高精度的setitimer)，构造一组函数，使得进程可以设置任一数目的计时器。

10.6 编写一段程序测试程序10-17中父进程和子进程的同步函数，要求进程创建一个文件并向文件写一个整数0，进程调用fork后父进程和子进程交替增加文件中的计数器的值，并打印哪一个进程(子进程或父进程)进行了增1操作。

10.7 在程序10-8中，若调用者捕捉了SIGABRT并从该信号处理程序中返回，为什么不是仅仅调用_exit，而要恢复缺省设置并再次调用kill？

10.8 为什么SVR4中的siginfo（见10.21节）在si_uid字段中传递实际的用户ID而非有效用户ID？

10.9 重写程序10-10，要求处理表10-1中的所有信号量，每次循环处理一个信号量而不是所有可能的信号量。

10.10 编写一段程序，要求在一个无限循环中调用 sleep(60)函数，每5分钟(即5次循环)取当前的日期和时间，并打印tm_sec域。将程序执行一晚上，请解释其结果。一些程序如 BSD中的cron守护进程是如何处理这类文件的？

10.11 修改程序3-3，要求：(a)将BUFFSIZE改为100；(b)用signal_intr函数捕捉SIGXFSZ信号量并打印消息，然后从信号量处理程序中返回； (c)如果没有写满请求的字节数，打印write的返回值。将软资源限制RLIMIT_FSIZE（见7.11节）变为1024字节(在shell中设置软资源限制，如果不行就直接在程序中的调用 setrlimit)，然后拷贝一个大于1024字节的文件，在各种不同的系统上运行新程序，其结果如何？为什么？

10.12 写一段调用fwrite的程序，要求使用一个较大的缓存区 (几个兆)，调用fwrite前调用alarm设置一秒钟以后调度信号量。在信号量处理程序中打印捕捉到的信号量然后返回。 fwrite可以完成吗？结果如何？