8.异常控制流

8.1 异常

异常是异常控制流的一种形式，他一部分由硬件实现，一部分由OS实现。
异常就是控制流中的突变，用来响应处理器状态中的某些变化。

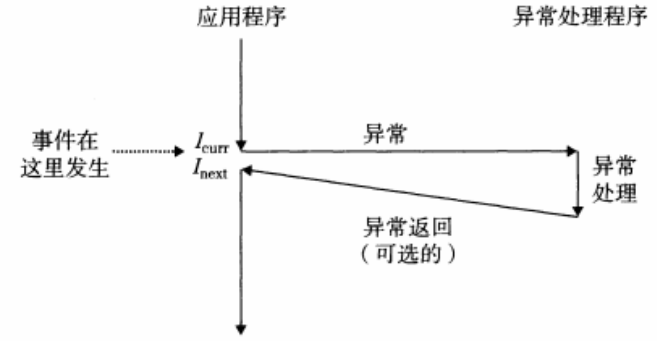


图 8-1 异常的剖析。处理器状态中的变化(事件)触发从应用程序到异常处理程序的突发的控制转移(异常)。在异常处理程序完成处理后，它将控制返回给被中断的程序或者终止

在任何情况下，当处理器检测到有事件发生时，他就会通过一张叫做异常表的跳转表，进行一个间接过程调用（异常），到一个专门设计用来处理这类事件的os子程序（异常处理程序）。当异常处理程序完成以后处理以后，根据引起异常的事件的类型，会发生一下三种情况中的一种：

- 处理程序将控制返回给当前指令Icuur
- 处理程序将控制返回给Inext
- 处理程序终止被中断的程序

8.1.1 异常处理

每种类型的异常都分配了一个唯一的非负整数的异常号。
其中一些号码是由处理器的设计者分配的（例如被零除、缺页、内存访问违例、断点以及算数运算溢出），其他号码由OS内核的设计者分配（包括系统调用和来自外部I/O设备的信号）

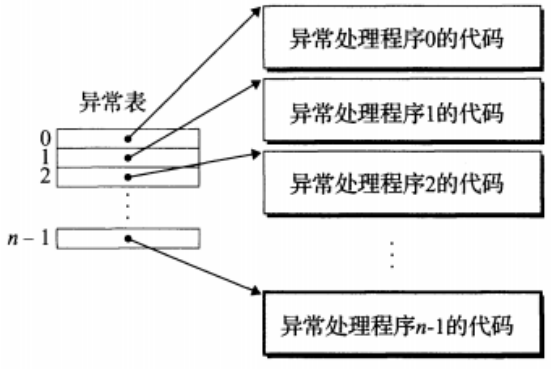


图 8-2 异常表。异常表是一张跳转表，其中表目 k 包含异常 k 的处理程序代码的地址

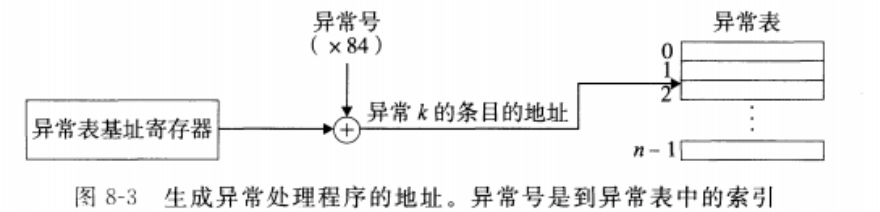


图 8-3 生成异常处理程序的地址。异常号是到异常表中的索引

异常类似于过程调用。不同之处在于异常要决定返回的方式、处理器会将一些额外的处理器状态压入栈中、如果控制从用户程序转移到内核，所有的这些项目会被压到内核栈中，而不是压到用户栈中、异常处理程序运行在内核模式下。

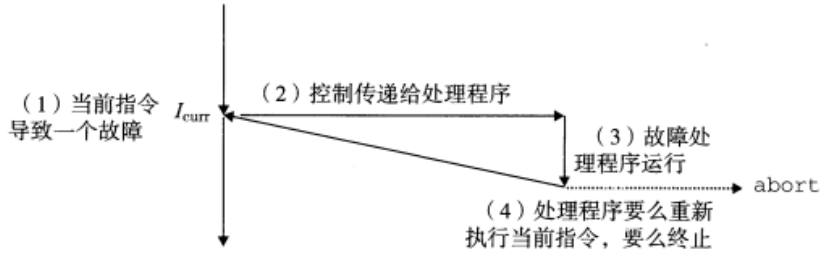
8.1.2 异常的类别

异常的可分为四类：中断、陷阱、故障和终止。
中断为I\O中断，陷阱为系统调用
同步是指由当前执行的指令造成的异常。

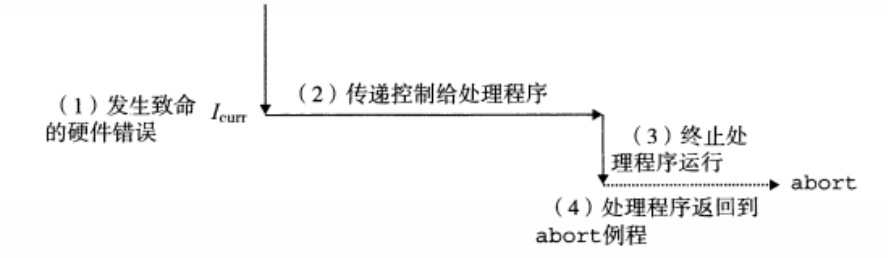
类别	原因	异步 / 同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

图 8-4 异常的类别。异步异常是由处理器外部的 I/O 设备中的事件产生的。同步异常是执行一条指令的直接产物

- 中断
  硬件中断的异常处理程序常常称为中断处理程序，它会返回到下一条指令。因为中断是在当前指令完成执行后，处理器注意到中断引脚的电压变高了，就从系统总线读取异常号，然后调用适当的中断处理程序。
- 陷阱和系统调用
  陷阱是有意的异常。陷阱处理程序总是将控制返回给下一条指令。陷阱最重要的用途时在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。
- 故障
  故障是由错误情况引起的，他可能能被故障处理程序修正。当故障发生时，处理器将控制转移给故障处理程序。如果程序能够修正这个错误情况，他就将控制返回到引起故障的指令，从而执行它。否则就返回到内核中的abort例程，abort例程会终止引起故障的应用程序。



  一个经典的故障是缺页异常。
- 终止
  终止是不可恢复的致命错误造成的结果，通常是一些硬件错误。比如DRAM或者SRAM位被损坏时发生的奇偶错误。



8.1.3 linux/x86-64系统中的异常

异常号	描述	异常类别
0	除法错误	故障
13	一般保护故障	故障
14	缺页	故障
18	机器检查	终止
32--255	操作系统定义的异常	中断或陷阱

图 8-9　x86-64 系统中的异常示例

1.LINUX/X86-64**故障和终止**

- 除法错误：除以0或者除法指令的结果对于目标操作数来说太大了。Unix不会试图从出发错误中恢复，而是选择终止程序。
- 一般保护故障：程序引用了一个未定义的虚拟内存区域，或者因为程序试图写一个只读的文本段。linux通常将一般保护故障报告为段故障
- 缺页：故障
- 机器检查：导致故障的指令执行中检测到致命的硬件错误时发生。

2.**系统调用**

在x86-64系统上，系统调用是通过一条称为syscall的陷阱指令来提供的。所有到linux系统调用的参数都是通过寄存器而不是栈传递。%rax包含系统调用号，寄存器%rdi、%rsi、%rdx、%r10、%r8和%r9包含最多6个参数。参数按照顺序摆放。从系统调用返回时，%rcx和%r11都会被破坏，%rax包含返回值。-4095到-1之间的负数返回值表明发生了错误，对应于负的error。

编号	名字	描述	编号	名字	描述
0	read	读文件	33	pause	挂起进程直到信号到达
1	write	写文件	37	alarm	调度告警信号的传送
2	open	打开文件	39	getpid	获得进程 ID
3	close	关闭文件	57	fork	创建进程
4	stat	获得文件信息	59	execve	执行一个程序
9	mmap	将内存页映射到文件	60	_exit	终止进程
12	brk	重置堆顶	61	wait4	等待一个进程终止
32	dup2	复制文件描述符	62	kill	发送信号到一个进程

图 8-10　Linux x86-64 系统中常用的系统调用示例

```
        First, call write(1, "hello, world\n", 13)
9       movq $1, %rax           write is system call 1
10      movq $1, %rdi           Arg1: stdout has descriptor 1
11      movq $string, %rsi      Arg2: hello world string
12      movq $len, %rdx         Arg3: string length
13      syscall                 Make the system call
```

8.2 进程

进程就是一个执行中程序的实例。每个程序都与运行在某个进程的上下文中。

进程提供给应用程序的关键抽象

- 一个独立的逻辑控制流，好像我们的程序在独占地使用寄存器
- 一个私有的地址空间，好像我们的程序独占地使用内存系统。

8.2.1 逻辑控制流

8.2.2 并发流

8.2.3 私有地址空间

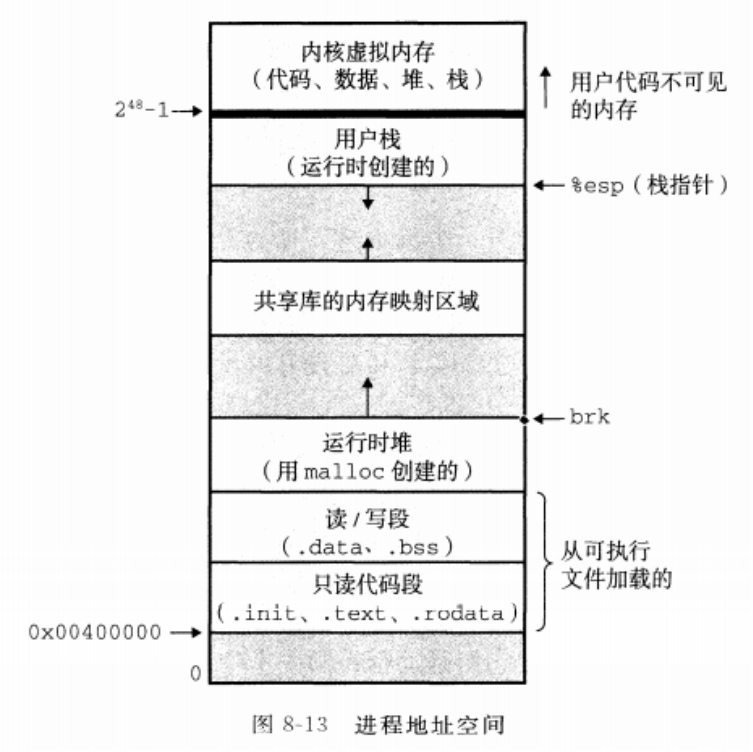


图 8-13　进程地址空间

8.2.4 用户模式和内核模式

处理器通常是用某个控制寄存器中的一个模式位来提供这种区分功能。设置了模式位，进程就运行在内核模式中。
/proc文件系统将许多内核数据结构的内容输出为一个用户程序可以读的文本文件的层次结构。

8.2.5 上下文切换

内核为每个进程维持一个上下文，上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值组成，包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、、状态寄存器、内核栈和各种内核数据结构，比如描述地址空间的页表，包含有关当前进程信息的进程表、以及包含进程已打开文件的信息的文件表。

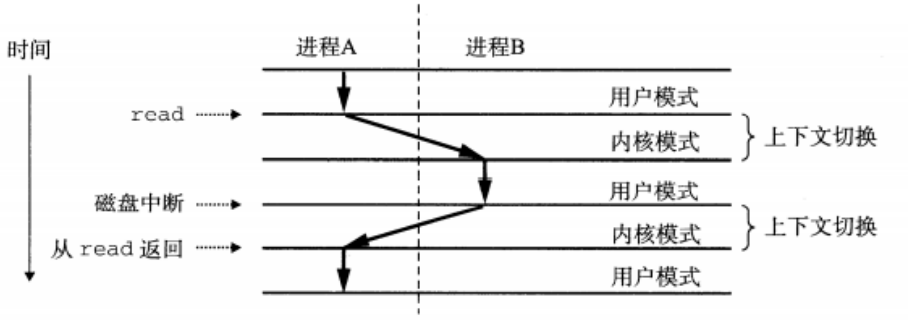


图 8-14　进程上下文切换的剖析

8.3 系统调用错误处理

8.4 进程控制

8.4.1 获取进程ID

每个进程都有一个唯一的正数进程ID（PID）。getpid函数返回调用进程的PID。getppid返回它的父进程的PID。

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

pid_t getpid(void);
pid_t getppid(void);
```

返回：调用者或其父进程的 PID。

getpid 和 getppid 函数返回一个类型为 pid_t 的整数值，在 Linux 系统上它在 types.h 中被定义为 int。

8.4.2 创建和终止进程

程序总是处于三种状态之一

- 运行：要么在CPU上执行，要么在等待被执行并最终会被内核调度
- 停止：进程的执行被挂起，且不会被调度。直到被唤醒
- 终止：进程永远的停止了。进程会因为三种原因终止：1.收到一个信号，该信号的默认行为是终止进程，2.从主程序返回，3.调用exit

```
#include <stdlib.h>

void exit(int status);
```

该函数不返回。

exit 函数以 status 退出状态来终止进程(另一种设置退出状态的方法是从主程序中返回一个整数值)。

fork（void）函数创建子进程。紫禁城的得到与父进程用户虚拟地址空间相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本。fork函数调用一次返回两次，在父进程中返回子进程的pid，在子进程中返回0。

8.4.3 回收子进程

当一个进程由于某种原因终止时，内核并不是立即把它从系统中清楚。进程被保持在一种已终止的状态中，直到被它的父进程回收。终止了还未被回收的进程称为僵死进程。
如果一个父进程终止了，内核会安排init进程成为他的孤儿进程的养父。init进程PID为1，是所有进程的祖先。
一个进程可以调用waitpid函数来等待他的子进程终止或者停止。

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

pid_t waitpid(pid_t pid, int *statusp, int options);
```

返回：如果成功，则为子进程的 PID，如果 WNOHANG，则为 0，如果其他错误，则为 −1。

1、判定等待集合的成员，pid》0，等待集合是一个单独的子进程；pid=-1，等待集合是由父进程所有的子进程组成的。
2.修改默认行为
将options设置为

- WNOHANG:如果等待集合中的任何子进程都还没有终止，那么就立即返回
- WUNTRACED:如果有进程被停止也会返回
- WCONTINUED:如果集合中的一个被停止的进程收到SIGCONT信号重新开始执行，也会返回。
- WNOHANG|WUNTRACED:立即返回，如果没有被暂停或是终止。

3.检查已回收子进程的退出状态

- WIFEXITED(status)：如果子进程通过调用 exit 或者一个返回(return)正常终止，就返回真。
- WEXITSTATUS(status)：返回一个正常终止的子进程的退出状态。只有在WIFEXITED()返回为真时，才会定义这个状态。
- WIFSIGNALED(status)：如果子进程是因为一个未被捕获的信号终止的，那么就返回真。
- WTERMSIG(status)：返回导致子进程终止的信号的编号。只有在 WIFSIGNALED()返回为真时，才定义这个状态。
- WIFSTOPPED(status)：如果引起返回的子进程当前是停止的，那么就返回真。
- WSTOPSIG(status)：返回引起子进程停止的信号的编号。只有在 WIFSTOPPED()返回为真时，才定义这个状态。
- WIFCONTINUED(status)：如果子进程收到 SIGCONT 信号重新启动，则返回真。

4.错误条件
如果调用进程没有子进程，那么waitpid返回-1，并设置errno为ECHILD。如果waitpid函数被一个信号中断，返回-1，设置errno为EINTR。

5.wait函数
是waitpid函数的简单版本

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

pid_t wait(int *statusp);
                                        返回：如果成功，则为子进程的 PID，如果出错，则为－1。
```

调用 wait(&status)等价于调用 waitpid(- 1,&status,0)。

6.案例
程序不会按照特定的顺序回收子进程。

8.4.4 让进程休眠

sleep 函数将一个进程挂起一段指定的时间。

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
                                        返回：还要休眠的秒数。
```

请求的时间量到了，sleep返回0，否则返回还剩下的休眠秒数。
pause（void）函数，让调用进程休眠，知道该进程收到一个信号。

8.4.5 加载并运行程序

execve函数在当前进程的上下文加载并运行一个新程序。

```
#include <unistd.h>

int execve(const char *filename, const char *argv[],
           const char *envp[]);
                                        如果成功，则不返回，如果错误，则返回－1。
```

argv变量指向一个以null结尾的指针数组，每个指针都指向一个参数字符串。argv[0]为可执行目标文件的名字，enpv为环境变量列表，每个指针指向一个“name=value‘的名字-指对。

8.5 信号

linux信号是一种更高层的软件形式的异常，它允许进程和内核中断其他进程。
一个信号就是一条小消息，他通知进程系统中发生了一个某种类型的事件。

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

8.5.1 信号术语

发送一个信号到目的进程是由两个不同步骤组成的：

- 发送信号：内核通过更新目的进程上下文中的某个状态，发送（递送）一个信号给目的进程。发送信号可以有如下两种原因：1.内核检测到一个系统事件，比如除0错误或者子进程终止。2.一个进程调用了kill函数，显式地要求内核发送一个信号给目的进程。一个进程可以发送信号给他自己
- 接收信号：当目的进程被内核强迫以某种方式对信号的发送做出反应时，他就接收了信号。
- 一个发出而没有被接收的信号叫做待处理信号，在任何时刻，一种类型至多只会有一个待处理信号。一个进程可以有选择性的阻塞接受某种信号。

8.5.2 发送信号

- 1.进程组
  每个进程都只属于一个进程组，进程组由一个正整数进程组ID来标识。getpgrp函数返回当前进程组ID。默认的，一个子进程和他的父进程同属一个进程组。一个进程可以使用set-pgid函数来改变自己或其它进程的进程组。

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
                                        返回：若成功则为0，若错误则为-1。
```

如果PID是0就是用当前进程的PID，如果PGID是0，就用pid指定的进程的PID作为进程组ID。

- 2.用/bin/kill程序发送信号

  /bin/kill程序可以向另外的进程发送任意信号。

  /bin/kill 程序可以向另外的进程发送任意的信号。比如，命令

  ```
  linux> /bin/kill -9 15213
  ```

  发送信号 9(SIGKILL)给进程 15213。一个为负的 PID 会导致信号被发送到进程组 PID 中的每个进程。比如，命令

  ```
  linux> /bin/kill -9 -15213
  ```

  发送一个 SIGKILL 信号给进程组 15213 中的每个进程。注意，在此我们使用完整路径/bin/kill，因为有些 Unix shell 有自己内置的 kill 命令。

- 3.从键盘发送信号

  用作业（job）这个抽象概念来表示对一条命令求值而创建的进程。在任何时刻，至多只有一个前台作业和0个或多个后台作业。shell为每个作业创建一个独立的进程组。进程组ID通常取自作业中父进程中的一个。

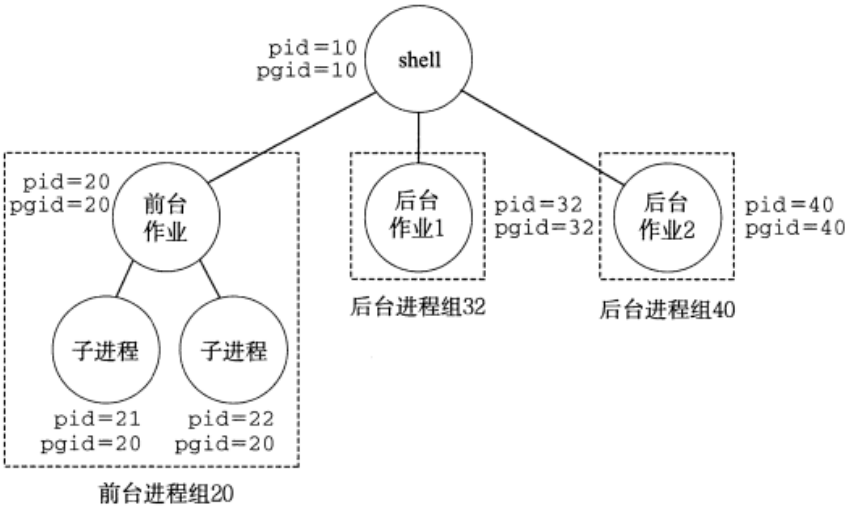


图 8-28 前台和后台进程组

  在键盘上输入Ctrl+C会导致内核发送一个SIGNIT信号到前台作业组中的每个进程，默认情况下是终止前台作业。Ctrl+Z会发送一个SIGSTP信号，挂起前台作业。

- 4.用kill函数发送信号

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

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

返回：若成功则为 0，若错误则为 −1。

  如果 pid 大于零，那么 kill 函数发送信号号码 sig 给进程 pid。如果 pid 等于零，那么 kill 发送信号 sig 给调用进程所在进程组中的每个进程，包括调用进程自己。如果 pid 小于零，kill 发送信号 sig 给进程组 |pid|(pid 的绝对值)中的每个进程。图 8-29 展示了一个示例，父进程用 kill 函数发送 SIGKILL 信号给它的子进程。

- 5.用alarm函数发送信号

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
                                    返回：前一次闹钟剩余的秒数，若以前没有设定闹钟，则为0。
```

alarm函数安排内核在secs秒后发送一个SIGALRM信号给调用进程。如果secs是零，那么不会调度安排新的闹钟(alarm)。在任何情况下，对alarm的调用都将取消任何待处理的(pending)闹钟，并且返回任何待处理的闹钟在被发送前还剩下的秒数(如果这次对alarm的调用没有取消它的话)；如果没有任何待处理的闹钟，就返回零。

8.5.3 接收信号

当进程p从内核模式切换到用户模式时，内核会检查进程p的未被阻塞的待处理信号的集合。如果集合非空，内核选择集合中的某个信号（通常是最小的k），并强制p接收信号k。

每个信号类型都有一个预定义的默认行文：

- 进程终止
- 进程终止并转存内存
- 进程停止（挂起）直到被SIHCONT信号重启
- 进程忽略该信号
  进程可以通过signal函数修改和信号相关联的默认行为。SIGNSTOP和SIGKILL是不能修改的。

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
                    返回：若成功则为指向前次处理程序的指针，若出错则为SIG_ERR(不设置errno)。
```

  hanlder参数：

- SIG_IGN，忽略类型为signum的信号
- SIG_DFL，类型为signum的信号行为恢复为默认行为
- handler为用户自定义的函数的地址。

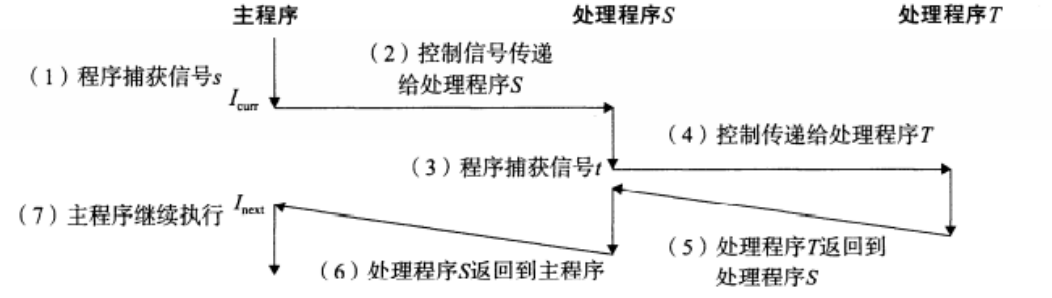


图 8-31 信号处理程序可以被其他信号处理程序中断

8.5.4 阻塞和解除阻塞信号

隐式阻塞机制：阻塞同类型
显式阻塞机制：利用sigprocmask函数和他的辅助函数，明确地阻塞和解除阻塞选定的信号。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

返回：如果成功则为0，若出错则为−1。

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

返回：若signum是set的成员则为1，如果不是则为0，若出错则为−1。

具体行为依赖于how的值

- SIG_BLOCK:把set中的信号添加到blocked中
- SIG_UNBLOCK:从blocked中删除set中的信号
- SIG_SETMASK:block=set

下属函数对set集合进行操作

- sigemptyset初始化set为空集合
- sigfillset把每个信号都添加到set中
- sigaddest把signum添加到set
- sigdelset从set中删除signum
- 如果signum是set地成员，sigismember返回1，否则返回0

8.5.5 编写信号处理程序

1.安全的信号处理

_Exit	fexecve	poll	sigqueue
_exit	fork	posix_trace_event	sigset
abort	fstat	pselect	sigsuspend
accept	fstatat	raise	sleep
access	fsync	read	sockatmark
aio_error	ftruncate	readlink	socket
aio_return	futimens	readlinkat	socketpair
aio_suspend	getegid	recv	stat
alarm	geteuid	recvfrom	symlink
bind	getgid	recvmsg	symlinkat
cfgetispeed	getgroups	rename	tcdrain
cfgetospeed	getpeername	renameat	tcflow
cfsetispeed	getpgrp	rmdir	tcflush
cfsetospeed	getpid	select	tcgetattr
chdir	getppid	sem_post	tcgetpgrp
chmod	getsockname	send	tcsendbreak
chown	getsockopt	sendmsg	tcsetattr
clock_gettime	getuid	sendto	tcsetpgrp
close	kill	setgid	time
connect	link	setpgid	timer_getoverrun
creat	linkat	setsid	timer_gettime
dup	listen	setsockopt	timer_settime
dup2	lseek	setuid	times
execl	lstat	shutdown	umask
execle	mkdir	sigaction	uname
execv	mkdirat	sigaddset	unlink
execve	mkfifo	sigdelset	unlinkat
faccessat	mkfifoat	sigemptyset	utime
fchmod	mknod	sigfillset	utimensat
fchmodat	mknodat	sigismember	utimes
fchown	open	signal	wait
fchownat	openat	sigpause	waitpid
fcntl	pause	sigpending	write
fdatasync	pipe	sigprocmask	

图 8.33 异步信号安全的函数(来源：man 7 signal。数据来自 Linux Foundation)

安全信号处理

- GO.处理程序要尽可能简单
- G1.在处理程序中只调用异步信号安全的函数
- G2.保存和恢复。把errno保存在一个局部变量中，在处理程序返回前回复他。只有在处理程序要返回时才有此必要。
- G3.访问共享全局数据结构时阻塞所有的信号，无论是主程序还是信号处理程序。
- G4.用volatile声明全局变量，告诉编译器不要缓存这个变量。
- G5.声明sig_atomic_t
  2.正确的信号处理

8.6 非本地跳转

将控制直接从一个函数转移到另一个正在执行的函数，而不需要经过正常的调用-返回序列。

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

返回：setjmp返回0，longjmp返回非零。

setjmp函数在env缓冲区中保存当前调用环境，以供后面的longjmp使用，并返回0。调用环境包括程序计数器、栈指针和通用目的寄存器。出于某种超出本书描述范围的原因，setjmp返回的值不能被赋值给变量：

```
rc = setjmp(env);  /* Wrong! */
```

不过它可以安全地用在switch或条件语句的测试中[62]。

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

从不返回。

longjmp函数从env缓冲区中恢复调用环境，然后触发一个从最近一次初始化env的setjmp调用的返回。然后setjmp返回，并带有非零的返回值retval。

通常哦那个与从一个深层嵌套的函数调用中立即返回，通常是检测到了某个错误。但可能会造成内存泄漏。