信号的概念

信号在我们的生活中随处可见， 如：古代战争中摔杯为号；现代战争中的信号弹；体育比赛中使用的信号枪.....他们都有共性：1. 简单 2. 不能携带大量信息 3. 满足某个特设条件才发送。

信号是信息的载体，Linux/UNIX 环境下，古老、经典的通信方式， 现下依然是主要的通信手段。

Unix 早期版本就提供了信号机制，但不可靠，信号可能丢失。Berkeley 和 AT&T 都对信号模型做了更改，增加了可靠信号机制。但彼此不兼容。POSIX.1 对可靠信号例程进行了标准化。

信号的机制

A 给 B 发送信号，B 收到信号之前执行自己的代码，收到信号后，不管执行到程序的什么位置，都要暂停运行，去处理信号，处理完毕再继续执行。与硬件中断类似——异步模式。但信号是软件层面上实现的中断，早期常被称为“软中断”。

信号的特质：由于信号是通过软件方法实现，其实现手段导致信号有很强的延时性。但对于用户来说，这个延迟时间非常短，不易察觉。

每个进程收到的所有信号，都是由内核负责发送的，内核处理。

与信号相关的事件和状态

产生信号:

1. 按键产生，如：Ctrl+c、Ctrl+z、Ctrl+\
2. 系统调用产生，如：kill、raise、abort
3. 软件条件产生，如：定时器 alarm
4. 硬件异常产生，如：非法访问内存(段错误)、除 0(浮点数例外)、内存对齐出错(总线错误)
5. 命令产生，如：kill 命令

递达：递送并且到达进程。

未决：产生和递达之间的状态。主要由于阻塞(屏蔽)导致该状态。

信号的处理方式:

1. 执行默认动作
2. 忽略(丢弃)
3. 捕捉(调用户处理函数)

Linux 内核的进程控制块 PCB 是一个结构体，task_struct, 除了包含进程 id，状态，工作目录，用户 id，组 id，文件描述符表，还包含了信号相关的信息，主要指阻塞信号集和未决信号集。

阻塞信号集(信号屏蔽字)： 将某些信号加入集合，对他们设置屏蔽，当屏蔽 x 信号后，再收到该信号，该信号的处理将推后(解除屏蔽后)

未决信号集:

1. 信号产生，未决信号集中描述该信号的位立刻翻转为 1，表信号处于未决状态。当信号被处理对应位翻转回为 0。这一时刻往往非常短暂。

2. 信号产生后由于某些原因(主要是阻塞)不能抵达。这类信号的集合称之为未决信号集。在屏蔽解除前，信号一直处于未决状态。

信号的编号

可以使用 kill –l 命令查看当前系统可使用的信号有哪些。

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

不存在编号为 0 的信号。其中 1-31 号信号称之为常规信号（也叫普通信号或标准信号），34-64 称之为实时信号，驱动编程与硬件相关。名字上区别不大。而前 32 个名字各不相同。

信号 4 要素

与变量三要素类似的，每个信号也有其必备 4 要素，分别是：

1. 编号 2. 名称 3. 事件 4. 默认处理动作

可通过 man 7 signal 查看帮助文档获取。也可查看/usr/src/linux-headers-3.16.0-30/arch/s390/include/uapi/asm/signal.h

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2

SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

在标准信号中，有一些信号是有三个“Value”，第一个值通常对 alpha 和 sparc 架构有效，中间值针对 x86、arm 和其他架构，最后一个应用于 mips 架构。一个‘-’表示在对应架构上尚未定义该信号。

不同的操作系统定义了不同的系统信号。因此有些信号出现在 Unix 系统内，也出现在 Linux 中，而有的信号出现在 FreeBSD 或 Mac OS 中却没有出现在 Linux 下。这里我们只研究 Linux 系统中的信号。

默认动作：

Term：终止进程

Ign： 忽略信号 (默认即时对该种信号忽略操作)

Core：终止进程，生成 Core 文件。(查验进程死亡原因， 用于 gdb 调试)

Stop：停止（暂停）进程

Cont：继续运行进程

注意从 man 7 signal 帮助文档中可看到 ：The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

这里特别强调了 **9) SIGKILL 和 19) SIGSTOP 信号，不允许忽略和捕捉，只能执行默认动作。甚至不能将其设置为阻塞。**

另外需清楚，只有每个信号所对应的事件发生了，该信号才会被递送(但不一定递达)，不应乱发信号！！

Linux 常规信号一览表

1) SIGHUP：当用户退出 shell 时，由该 shell 启动的所有进程将收到这个信号，默认动作为终止进程

2) SIGINT：当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程。

3) SIGQUIT：当用户按下<ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号。默认动作为终止进程。

4) SIGILL：CPU 检测到某进程执行了非法指令。默认动作为终止进程并产生 core 文件

5) SIGTRAP：该信号由断点指令或其他 trap 指令产生。默认动作为终止里程 并产生 core 文件。

6) SIGABRT：调用 abort 函数时产生该信号。默认动作为终止进程并产生 core 文件。

7) SIGBUS：非法访问内存地址，包括内存对齐出错，默认动作为终止进程并产生 core 文件。

8) SIGFPE：在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等所有的算法错误。默认动作为终止进程并产生 core 文件。

9) SIGKILL：无条件终止进程。本信号不能被忽略，处理和阻塞。默认动作为终止进程。它向系统管理员提供了可以杀死任何进程的方法。

10) SIGUSE1：用户定义 的信号。即程序员可以在程序中定义并使用该信号。默认动作为终止进程。

11) SIGSEGV：指示进程进行了无效内存访问。默认动作为终止进程并产生 core 文件。

12) SIGUSR2：另外一个用户自定义信号，程序员可以在程序中定义并使用该信号。默认动作为终止进程。

13) SIGPIPE：Broken pipe 向一个没有读端的管道写数据。默认动作为终止进程。

14) SIGALRM：定时器超时，超时的时间 由系统调用 alarm 设置。默认动作为终止进程。

15) SIGTERM：程序结束信号，与 SIGKILL 不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行 shell 命令 Kill 时，缺省产生这个信号。默认动作为终止进程。

16) SIGSTKFLT：Linux 早期版本出现的信号，现仍保留向后兼容。默认动作为终止进程。

17) SIGCHLD：子进程状态发生变化时，父进程会收到这个信号。默认动作为**忽略**这个信号。

18) SIGCONT：如果进程已停止，则使其继续运行。默认动作为继续/忽略。

19) SIGSTOP：停止进程的执行。信号不能被忽略，处理和阻塞。默认动作为暂停进程。

20) SIGTSTP：停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号。默认动作为暂停进程。

21) SIGTTIN：后台进程读终端控制台。默认动作为暂停进程。

22) SIGTTOU：该信号类似于 SIGTTIN，在后台进程要向终端输出数据时发生。默认动作为暂停进程。

23) SIGURG：套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达，默认动作为忽略该信号。

24) SIGXCPU：进程执行时间超过了分配给该进程的 CPU 时间 ，系统产生该信号并发送给该进程。默认动作为终止进程。

25) SIGXFSZ：超过文件的最大长度设置。默认动作为终止进程。

26) SIGVTALRM：虚拟时钟超时时产生该信号。类似于 SIGALRM，但是该信号只计算该进程占用 CPU 的使用时间。默认动作为终止进程。

27) SGIPROF：类似于 SIGVTALRM，它不公包括该进程占用 CPU 时间还包括执行系统调用时间。默认动作为终止进程。

28) SIGWINCH：窗口变化大小时发出。默认动作为忽略该信号。

29) SIGIO：此信号向进程指示发出了一个异步 IO 事件。默认动作为忽略。

30) SIGPWR：关机。默认动作为终止进程。

31) SIGSYS：无效的系统调用。默认动作为终止进程并产生 core 文件。

34) SIGRTMIN ～ (64) SIGRTMAX：LINUX 的实时信号，它们没有固定的含义（可以由用户自定义）。所有的实时信号的默认动作都为终止进程。

信号的产生

终端按键产生信号

Ctrl + c → 2) SIGINT（终止/中断） "INT" ----Interrupt

Ctrl + z → 20) SIGTSTP（暂停/停止） "T" ----Terminal 终端。

Ctrl + \ → 3) SIGQUIT（退出）

硬件异常产生信号

除 0 操作 → 8) SIGFPE (浮点数例外) "F" -----float 浮点数。

非法访问内存 → 11) SIGSEGV (段错误)

总线错误 → 7) SIGBUS

kill 函数/命令产生信号

kill 命令产生信号：kill -SIGKILL pid

kill 函数：给指定进程发送指定信号(不一定杀死)

int kill(pid_t pid, int sig); 成功：0；失败：-1 (ID 非法，信号非法，普通用户杀 init 进程等权级问题)，设置 errno

sig：不推荐直接使用数字，应使用宏名，因为不同操作系统信号编号可能不同，但名称一致。

pid > 0: 发送信号给指定的进程。

pid = 0: 发送信号给 与调用 kill 函数进程属于同一进程组的所有进程。

pid < 0: 取|pid|发给对应进程组。

pid = -1：发送给进程有权限发送的系统中所有进程。

进程组：每个进程都属于一个进程组，进程组是一个或多个进程集合，他们相互关联，共同完成一个实体任务，每个进程组都有一个进程组长，默认进程组 ID 与进程组长 ID 相同。

权限保护：super 用户(root)可以发送信号给任意用户，普通用户是不能向系统用户发送信号的。 kill -9 (root 用户的 pid) 是不可以的。同样，普通用户也不能向其他普通用户发送信号，终止其进程。 只能向自己创建的进程发送信号。普通用户基本规则是：发送者实际或有效用户 ID == 接收者实际或有效用户 ID

练习：循环创建 5 个子进程，父进程用 kill 函数终止任一子进程。 【kill.c】

软件条件产生信号

alarm 函数

设置定时器(闹钟)。在指定 seconds 后，内核会给当前进程发送 14）SIGALRM 信号。进程收到该信号，默认动作终止。

每个进程都有且只有唯一个定时器。

unsigned int alarm(unsigned int seconds); 返回 0 或剩余的秒数，无失败。

常用：取消定时器 alarm(0)，返回旧闹钟余下秒数。

例：alarm(5) → 3sec → alarm(4) → 5sec → alarm(5) → alarm(0)

定时，与进程状态无关(自然定时法)！就绪、运行、挂起(阻塞、暂停)、终止、僵尸...无论进程处于何种状态，alarm 都计时。

练习：编写程序，测试你使用的计算机 1 秒钟能数多少个数。 【alarm .c】

使用 time 命令查看程序执行的时间。 程序运行的瓶颈在于 IO，优化程序，首选优化 IO。

实际执行时间 = 系统时间 + 用户时间 + 等待时间

setitimer 函数

设置定时器(闹钟)。 可代替 alarm 函数。精度微秒 us，可以实现周期定时。

int setitimer(int which, const struct itimerval *new_value, struct itimerval *old_value); 成功：0；失败：-1，设置 errno

参数：which：指定定时方式

① 自然定时：ITIMER_REAL → 14）SIGLARM　　计算自然时间

② 虚拟空间计时(用户空间)：ITIMER_VIRTUAL → 26）SIGVTALRM　　只计算进程占用 cpu 的时间

③ 运行时计时(用户+内核)：ITIMER_PROF → 27）SIGPROF　　计算占用 cpu 及执行系统调用的时间

练习：使用 setitimer 函数实现 alarm 函数，重复计算机 1 秒数数程序。　　【setitimer_alarm.c】

拓展练习，结合 man page 编写程序，测试 it_interval、it_value 这两个参数的作用。　　【setitimer_cycle.c】

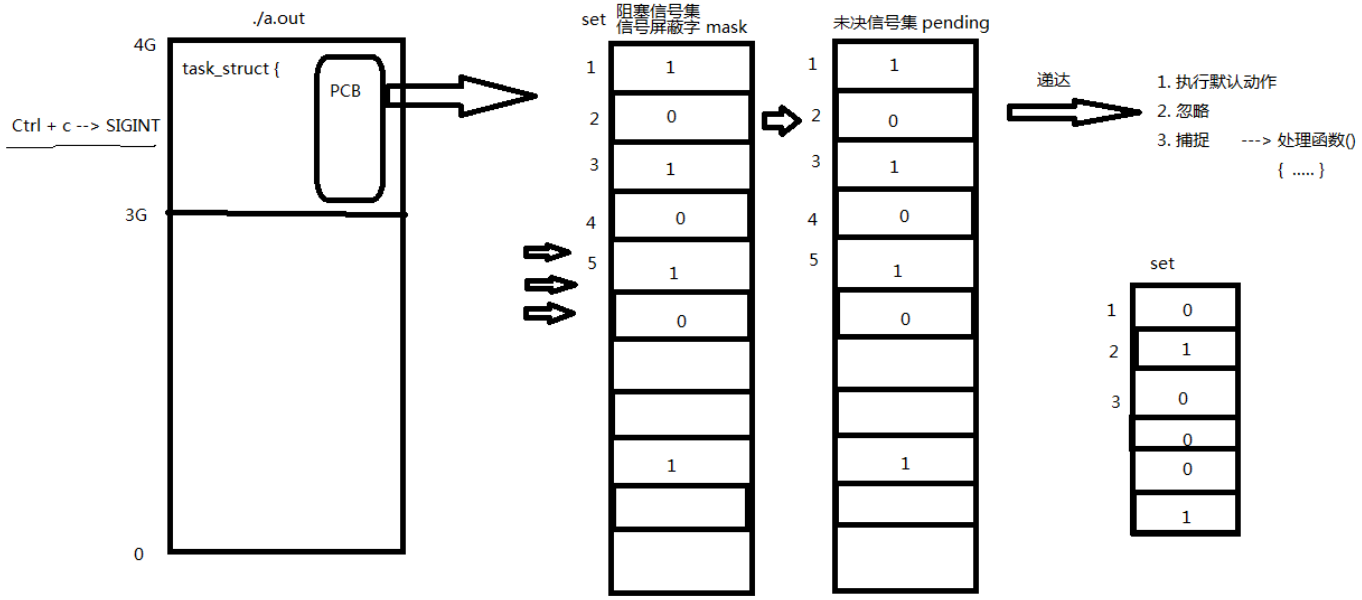
提示：　it_interval：用来设定两次定时任务之间间隔的时间。

it_value：定时的时长

两个参数都设置为 0，即清 0 操作。

信号集操作函数

内核通过读取未决信号集来判断信号是否应被处理。信号屏蔽字 mask 可以影响未决信号集。而我们可以在应用程序中自定义 set 来改变 mask。已达到屏蔽指定信号的目的。



信号集设定

sigset_t　set;　　// typedef unsigned long sigset_t;

int sigemptyset(sigset_t *set);　　将某个信号集清 0　　成功：0；失败：-1

int sigfillset(sigset_t *set);　　将某个信号集置 1　　成功：0；失败：-1

int sigaddset(sigset_t *set, int signum);　　将某个信号加入信号集　　成功：0；失败：-1

int sigdelset(sigset_t *set, int signum);　　将某个信号清出信号集　　成功：0；失败：-1

int sigismember(const sigset_t *set, int signum);判断某个信号是否在信号集中　返回值：在集合：1；不在：0；出错：-1

sigset_t 类型的本质是位图。但不应该直接使用位操作，而应该使用上述函数，保证跨系统操作有效。

对比认知 select 函数。

sigprocmask 函数

用来屏蔽信号、解除屏蔽也使用该函数。其本质，读取或修改进程的信号屏蔽字(PCB 中)

严格注意，屏蔽信号：只是将信号处理延后执行(延至解除屏蔽)；而忽略表示将信号丢处理。

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); 成功：0；失败：-1，设置 errno

参数：

set：传入参数，是一个位图，set 中哪位置 1，就表示当前进程屏蔽哪个信号。

oldset：传出参数，保存旧的信号屏蔽集。

how 参数取值： 假设当前的信号屏蔽字为 mask

1. SIG_BLOCK: 当 how 设置为此值，set 表示需要屏蔽的信号。相当于 mask = mask|set
2. SIG_UNBLOCK: 当 how 设置为此，set 表示需要解除屏蔽的信号。相当于 mask = mask & ~set
3. SIG_SETMASK: 当 how 设置为此，set 表示用于替代原始屏蔽及的新屏蔽集。相当于 mask = set 若，调用 sigprocmask 解除了对当前若干个信号的阻塞，则在 sigprocmask 返回前，至少将其中一个信号递达。

sigpending 函数

读取当前进程的**未决**信号集

int sigpending(sigset_t *set); set 传出参数。 返回值：成功：0；失败：-1，设置 errno

练习：编写程序。把所有常规信号的未决状态打印至屏幕。 【sigpending.c】

信号捕捉

signal 函数

注册一个信号捕捉函数：

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

该函数由 ANSI 定义，由于历史原因在不同版本的 Unix 和不同版本的 Linux 中可能有不同的行为。因此应该尽量避免使用它，取而代之使用 sigaction 函数。

void (*signal(int signum, void (*sighandler_t)(int))) (int);

能看出这个函数代表什么意思吗？ 注意多在复杂结构中使用 typedef。

sigaction 函数

修改信号处理动作（通常在 Linux 用其来注册一个信号的捕捉函数）

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);　成功：0；失败：-1，设置 errno

参数：

act：传入参数，新的处理方式。

oldact：传出参数，旧的处理方式。　【signal.c】

struct sigaction 结构体

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

sa_restorer：该元素是过时的，不应该使用，POSIX.1 标准将不指定该元素。(弃用)

sa_sigaction：当 sa_flags 被指定为 SA_SIGINFO 标志时，使用该信号处理程序。(很少使用)

重点掌握：

① sa_handler：指定信号捕捉后的处理函数名(即注册函数)。也可赋值为 SIG_IGN 表忽略 或 SIG_DFL 表执行默认动作

② sa_mask: 调用信号处理函数时，所要屏蔽的信号集合(信号屏蔽字)。注意：仅在处理函数被调用期间屏蔽生效，是临时性设置。

③ sa_flags：通常设置为 0，表使用默认属性。

信号捕捉特性

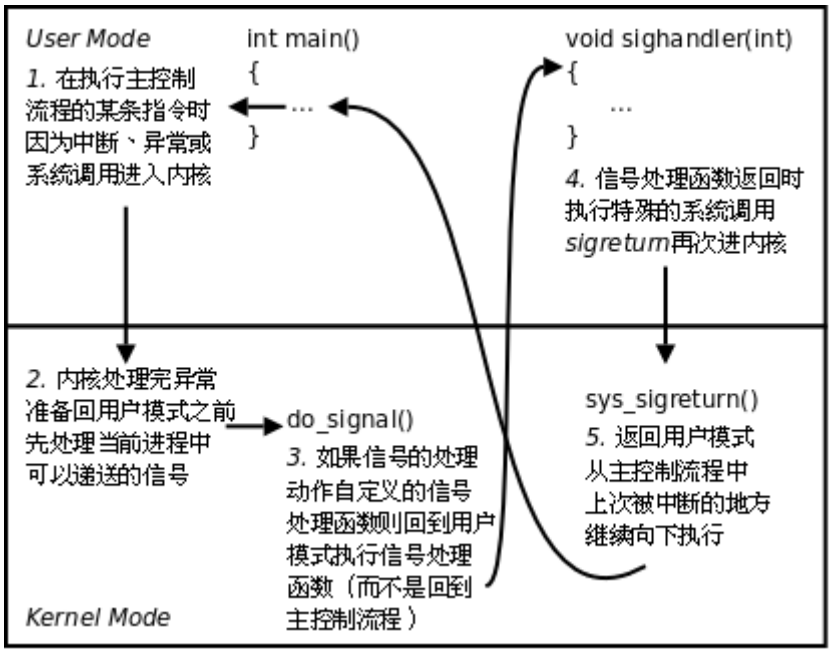
1. 进程正常运行时，默认 PCB 中有一个信号屏蔽字，假定为☆，它决定了进程自动屏蔽哪些信号。当注册了某个信号捕捉函数，捕捉到该信号以后，要调用该函数。而该函数有可能执行很长时间，在这期间所屏蔽的信号不由☆来指定。而是用 sa_mask 来指定。调用完信号处理函数，再恢复为☆。
2. XXX 信号捕捉函数执行期间，XXX 信号自动被屏蔽。
3. 阻塞的常规信号不支持排队，产生多次只记录一次。（后 32 个实时信号支持排队）

练习 1：为某个信号设置捕捉函数　【sigaction1.c】

练习 2：验证在信号处理函数执行期间，该信号多次递送，那么只在处理函数之行结束后，处理一次。【sigaction2.c】

练习 3：验证 sa_mask 在捕捉函数执行期间的屏蔽作用。　【sigaction3.c】

内核实现信号捕捉过程：



SIGCHLD 信号

SIGCHLD 的产生条件

子进程终止时

子进程接收到 SIGSTOP 信号停止时

子进程处在停止态，接受到 SIGCONT 后唤醒时

借助 SIGCHLD 信号回收子进程

子进程结束运行，其父进程会收到 SIGCHLD 信号。该信号的默认处理动作是忽略。可以捕捉该信号，在捕捉函数中完成子进程状态的回收。

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

void sys_err(char *str)
{
	perror(str);
	exit(1);
}
void do_sig_child(int signo)
```

```
{
    int status;     pid_t pid;
    while ((pid = waitpid(0, &status, WNOHANG)) > 0) {
        if (WIFEXITED(status))
            printf("child %d exit %d\n", pid, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("child %d cancel signal %d\n", pid, WTERMSIG(status));
    }
}
int main(void)
{
    pid_t pid;      int i;
    for (i = 0; i < 10; i++) {
        if ((pid = fork()) == 0)
            break;
        else if (pid < 0)
            sys_err("fork");
    }
    if (pid == 0) {
        int n = 1;
        while (n--) {
            printf("child ID %d\n", getpid());
            sleep(1);
        }
        return i+1;
    } else if (pid > 0) {
        struct sigaction act;
        act.sa_handler = do_sig_child;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        sigaction(SIGCHLD, &act, NULL);

        while (1) {
            printf("Parent ID %d\n", getpid());
            sleep(1);
        }
    }
    return 0;
}
```

分析该例子。结合 17)SIGCHLD 信号默认动作，掌握父使用捕捉函数回收子进程的方式。 【sigchild.c】

如果子进程主逻辑中 sleep(1)可以吗？可不可以将程序中，捕捉函数内部的 while 替换为 if？为什么？

if ((pid = waitpid(0, &status, WNOHANG)) > 0) { ... }

思考：信号不支持排队，当正在执行 SIGCHLD 捕捉函数时，再过来一个或多个 SIGCHLD 信号怎么办？

子进程结束 status 处理方式

pid_t waitpid(pid_t pid, int *status, int options)

options：

WNOHANG

没有子进程结束，立即返回

WUNTRACED

如果子进程由于被停止产生的 SIGCHLD，waitpid 则立即返回

WCONTINUED

如果子进程由于被 SIGCONT 唤醒而产生的 SIGCHLD，waitpid 则立即返回

status：

WIFEXITED(status)

子进程正常 exit 终止，返回真

WEXITSTATUS(status)返回子进程正常退出值

WIFSIGNALED(status)

子进程被信号终止，返回真

WTERMSIG(status)返回终止子进程的信号值

WIFSTOPPED(status)

子进程被停止，返回真

WSTOPSIG(status)返回停止子进程的信号值

WIFCONTINUED(status)

SIGCHLD 信号注意问题

1. 子进程继承父进程的信号屏蔽字和信号处理动作，但子进程没有继承未决信号集 spending。
2. 注意注册信号捕捉函数的位置。
3. 应该在 fork 之前，阻塞 SIGCHLD 信号。注册完捕捉函数后解除阻塞。

中断系统调用

系统调用可分为两类：慢速系统调用和其他系统调用。

1. 慢速系统调用：可能会使进程永远阻塞的一类。如果在阻塞期间收到一个信号，该系统调用就被中断,不再继续执行(早期)；也可以设定系统调用是否重启。如，read、write、pause、wait...
2. 其他系统调用：getpid、getppid、fork...

结合 pause，回顾慢速系统调用：

慢速系统调用被中断的相关行为，实际上就是 pause 的行为： 如，read

① 想中断 pause，信号不能被屏蔽。

② 信号的处理方式必须是捕捉 (默认、忽略都不可以)

③ 中断后返回-1， 设置 errno 为 EINTR(表“被信号中断”)

可修改 sa_flags 参数来设置被信号中断后系统调用是否重启。SA_INTERRURT 不重启。 SA_RESTART 重启。

扩展了解：

sa_flags 还有很多可选参数，适用于不同情况。如：捕捉到信号后，在执行捕捉函数期间，不希望自动阻塞该信号，可将 sa_flags 设置为 SA_NODEFER，除非 sa_mask 中包含该信号。