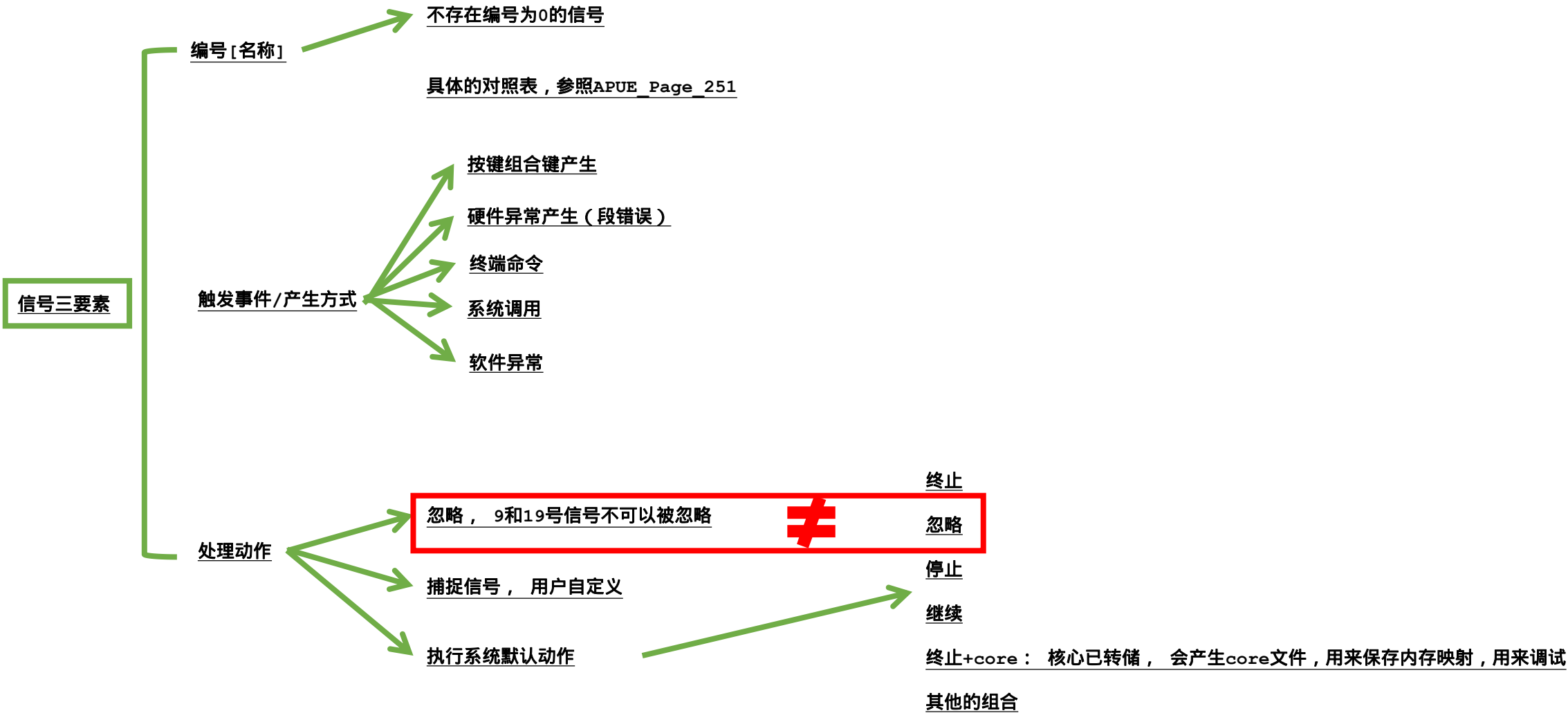


信号概念

1. 信号是一种软中断， 提供了一种处理异步事件的方法。



函数_signal

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

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

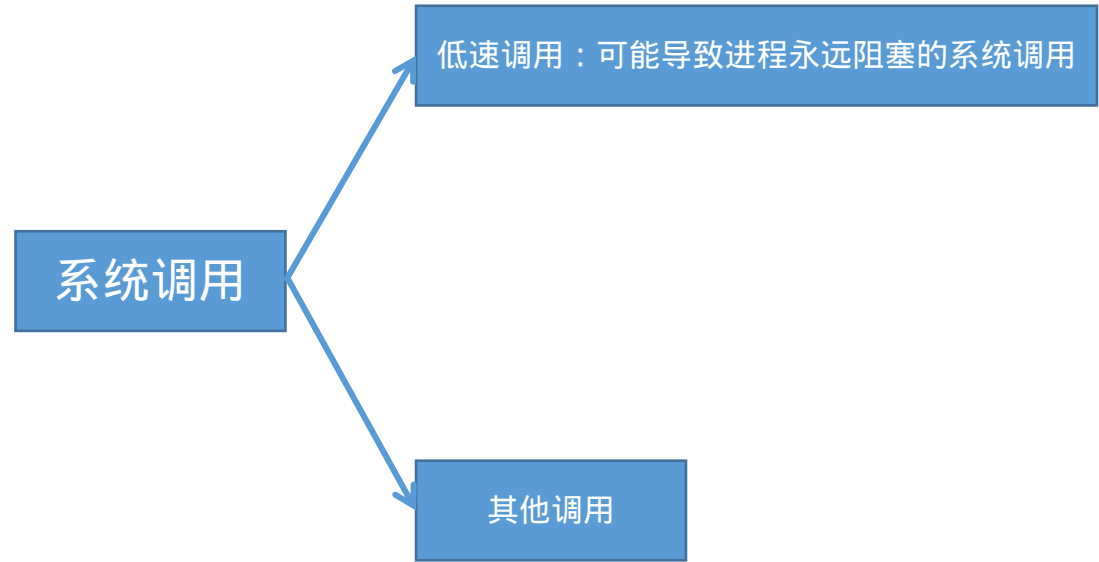
成功，返回以前的信号处理配置，失败，返回SIG_ERR

handler:	SIG_IGN, SIG_DFL, 函数指针
	SIG_IGN: 忽略
	SIG_DFL: 系统默认动作
	函数指针: 调用该函数

子进程继承父进程的信号处理方式，调用exec后，将原先设置为用户自定义函数捕捉方式的信号都改为默认动作

signal默认重启被信号中断的系统调用，且在捕捉的过程中，自动屏蔽自身信号

中断的系统调用



1. 当一个低速调用被阻塞期间被一个信号中断，那么该调用将不再继续执行，返回出错，并设置`errno`为`EINTR`

2. 我们可以手动重启该被信号中断的低速系统调用

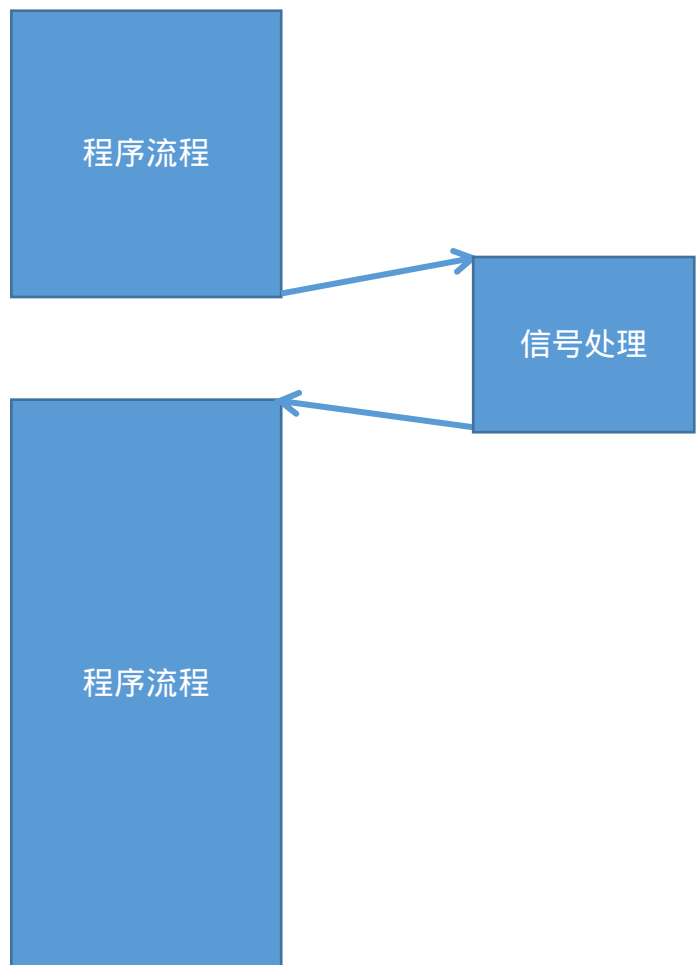
```
again:
    if((n = read(fd, buf, BUFSIZE)) < 0)
    {
        if(errno == EINTR)
            goto again;
    }
```

3. 4.2BSD提供了某些被中断系统调用的自动重启动功能，POSIX.1要求只有中断信号的`SA_RESTART`标志被设置，实现才会重启系统调用

`sigaction`中可以设置

linux下，`signal`默认是重启被中断的系统调用的

可重入函数



1. 异步信号不安全函数

当本身在程序流程中进行的一个函数因为信号的到来而被临时中断，然后因为信号处理流程中调用了一个结束状态不明的函数，从而导致的问题。本质是对一个公共资源的重复不完整的访问。

SIGCHLD信号

- 1. 子进程状态改变后产生此信号， 递送给自己的父进程
- 2. 父进程处理方式
 - 2.1 SIG IGN：忽略该信号，但是子进程不会变成僵尸进程
 - 2.2 SIG DFL：采用默认动作， 默认动作为忽略，子进程变为僵尸进程
 - 2.3 捕捉： 调用对应的处理函数

远古不可靠的信号

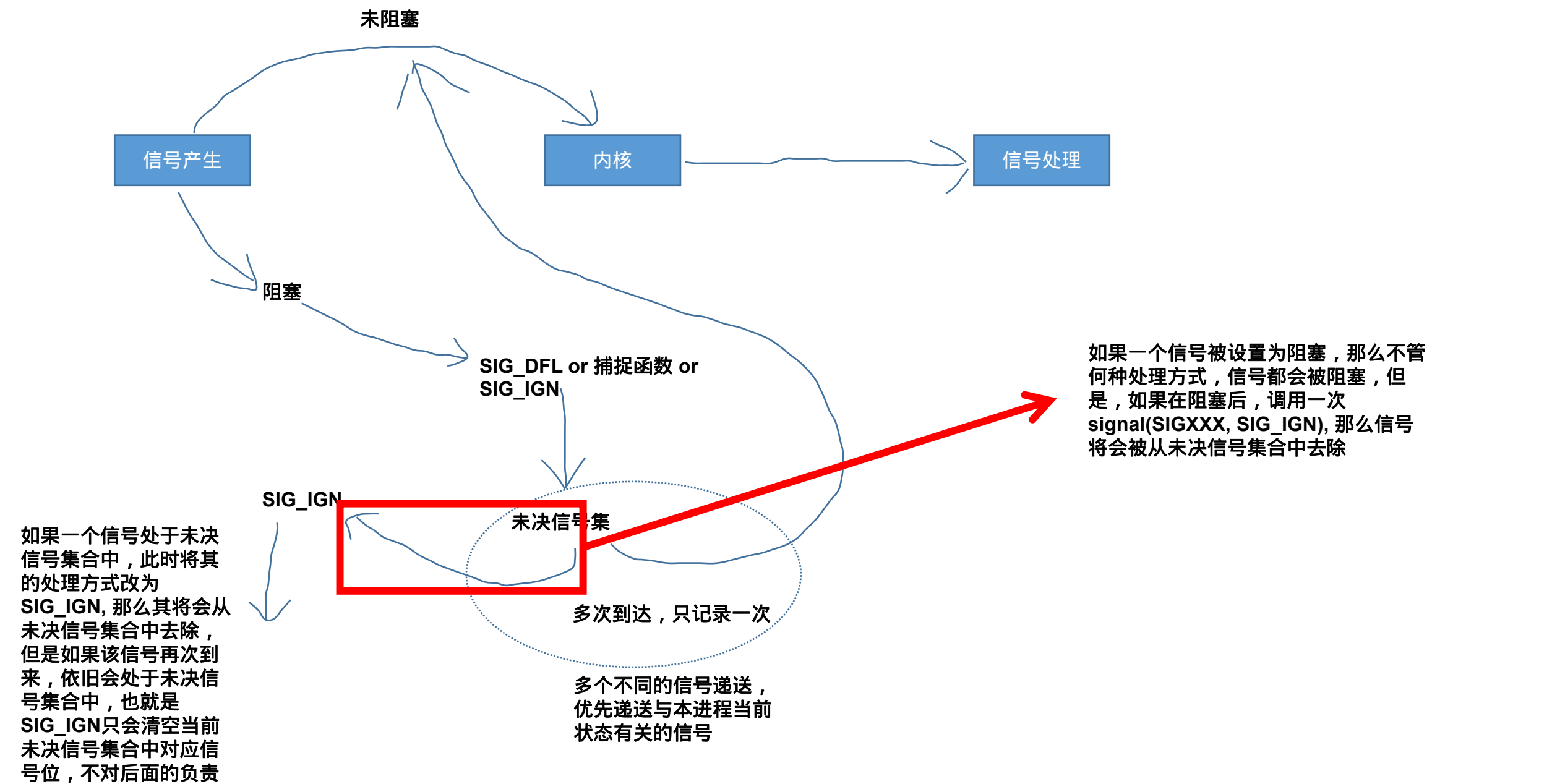
1. 信号可能会丢失[一个信号发生了，但是进程可能不知道]

在早期的信号机制中，当进程进入信号处理程序的时候，随机就把信号的动作重置为默认行为

2. 对信号的控制能力差，能够捕捉和忽略信号，但是不能阻塞信号

我们在有的时候，不想对信号进行处理，但是想保留其到达的痕迹，就需要一个可以临时阻塞，到时释放的信号处理手段

现代可靠的信号



信号相关函数_kill / raise

```
1. int kill(pid_t pid, int signo)
2. int raise(int signo)
//成功，返回0， 失败， 返回-1
```

- pid > 0 : 将该信号发送给有权限发送的对应pid的进程
- pid == 0: 将该信号发送给有权限发送的和调用进程属于同一进程组的所有进程。
- pid < 0 : 将该信号发送给有权限发送的进程组ID等于pid绝对值的进程
- pid == -1: 将该信号发送给所有有权限发送的进程

信号发送的权限问题：基本规则是发送者的实际用户ID或者有效用户ID等于接受者的实际用户ID或有效用户ID

当kill的参数signo为0 的时候，kill仍正常执行，但是不发送信号， 仍然执行错误检查， 可以用来验证一个特定的进程是否存在：
如果进程存在，返回0.
如果进程不存在，返回-1，并设置errno为ESRCH

如果kill为调用进程发送信号，而且信号没有被阻塞，那么在kill返回之前， 信号将被内核接受

信号相关函数_alarm/pause

`unsigned int alarm(unsigned int seconds)`

返回0或者以前定时剩余的秒数

一个进程只能有一个闹钟时间，新的值会替代旧的时钟值，并将旧的剩余值返回

`int pause(void)`

使调用进程挂起直到捕捉到一个信号

返回-1，errno设置为EINTR

只有执行了一个信号处理程序并返回，pause才返回

自己实现sleep函数

```
unsigned int sleep_1(unsigned int seconds)
{
    sighandler fp = signal(SIGALRM, sig_alm);
    if(fp == SIG_ERR)
        return seconds;
    alarm(seconds);
    pause();
    signal(SIGALRM, fp);
    return (alarm(0));
}
```

竞态条件：解决方法

1. setjmp

2. sigprocmask 和 sigsuspend

信号相关函数_alarm/sleep

```
unsigned int sleep_1(unsigned int seconds)
{
    sighandler fp = signal(SIGALRM, sig_alm);
    if(fp == SIG_ERR)
        return seconds;
    alarm(seconds);
    pause();
    signal(SIGALRM, fp);
    return (alarm(0));
}
```

竞态条件：解决方法

1. setjmp
2. sigprocmask 和 sigsuspend

```
static jmp_buf env_alm;

static void sig_alm(int signo)
{
    longjmp(env_alm, 1);
}

unsigned int sleep2(unsigned int seconds)
{
    if(signal(SIGALRM, sig_alm) == SIG_ERR)
        return seconds;
    if(setjmp(env_alm) == 0)
    {
        alarm(seconds);
        pause();
    }
    return alarm(0);
}
```

如果在调用pause前收到信号，那么pause就不会被执行，但是在涉及与其他信号交互的时候，会中断其他的信号用户处理程序。

信号集

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

```
int sigemptyset(sigset_t *set);
```

```
int sigfillset(sigset_t *set);
```

```
int sigaddset(sigset_t *set, int signum);
```

```
int sigdelset(sigset_t *set, int signum);
```

succ, return 0, fail, return -1

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

ret true or false

信号相关函数_sigprocmask

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

succ, return 0, fail, return -1

设置当前进程的信号屏蔽字，并返回旧的屏蔽字，
如果set为空指针，则不改变信号屏蔽字，how也无意义

The behavior of the call is dependent on the value of how, as follows.

SIG_BLOCK

The set of blocked signals is the union of the current set and the set argument.

SIG_UNBLOCK

The signals in set are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.

SIG_SETMASK

The set of blocked signals is set to the argument set.

调用该函数后，如果有任何未决的信号不在被阻塞，则在该函数返回之前，至少将其中之一递送给该进程

信号相关函数_sigpending

```
int sigpending(sigset_t *set);
```

```
succ, return 0, fail, return -1
```

获取当前进程的未决信号集

调用该函数后，如果有任何未决的信号不再被阻塞，则在该函数返回之前，至少将其中之一递送给该进程

信号相关函数_sigaction

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

succ, return 0, fail, return -1

检查和修改与指定信号相关联的处理动作

if act == NULL: return just return the old sigaction

else: set the act to the sigaction and return the old

```
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_handler: SIG_IGN、SIG_DFL、function address, 当设置为function address的时候, sa_mask设置为一个信号集, 在调用该信号捕捉函数之前, 将会把该屏蔽集加入到信号屏蔽集合中, 当函数结束的时候, 会去掉, 同时, 该屏蔽集合中会自动包含激活自身的一个信号。

sa_flags:指定了对信号进行处理的选项

sa_sigaction: 在sa_flags设置为SA_SIGINFO 时, 系统采用此处的函数作为信号捕捉函数, 可以用来进行信号的数据传递

信号相关函数_sigaction

sa_flags参数：

sigaction默认不重启被信号中断的系统调用

SA_SIGINFO (since Linux 2.2)

The signal handler takes three arguments, not one. In this case, sa_sigaction should be set instead of sa_handler. This flag is meaningful only when establishing a signal handler.

SA_RESTART

Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals. This flag is meaningful only when establishing a signal handler. See **signal(7)** for a discussion of system call restarting.

SA_NOCLDWAIT (since Linux 2.6)

If signum is **SIGCHLD**, do not transform children into zombies when they terminate. See also **waitpid(2)**. This flag is meaningful only when establishing a handler for **SIGCHLD**, or when setting that signal's disposition to **SIG_DFL**.

If the **SA_NOCLDWAIT** flag is set when establishing a handler for **SIGCHLD**, POSIX.1 leaves it unspecified whether a **SIGCHLD** signal is generated when a child process terminates. On Linux, a **SIGCHLD** signal is generated in this case; on some other implementations, it is not.

SA_SIGINFO (since Linux 2.2)

The signal handler takes three arguments, not one. In this case, sa_sigaction should be set instead of sa_handler. This flag is meaningful only when establishing a signal handler.

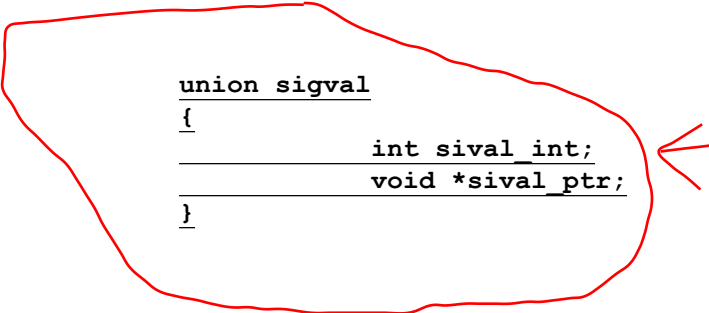
SA_NOCLDSTOP

If signum is **SIGCHLD**, do not receive notification when child processes stop (i.e., when they receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU**) or resume (i.e., they receive **SIGCONT**) (see **wait(2)**). This flag is meaningful only when establishing a handler for **SIGCHLD**.

信号相关函数_sigaction

void (*sa_sigaction)(int, siginfo_t *, void *)

参数siginfo_t, 详细的包括信号的一些附带信息



```
union sigval
{
    int sival_int;
    void *sival_ptr;
}
```

```
siginfo_t {
    int      si_signo;    /* Signal number */
    int      si_errno;    /* An errno value */
    int      si_code;     /* Signal code */
    int      si_trapno;   /* Trap number that caused
                           hardware-generated signal
                           (unused on most architectures) */

    pid_t    si_pid;      /* Sending process ID */
    uid_t    si_uid;      /* Real user ID of sending process */
    int      si_status;    /* Exit value or signal */
    clock_t  si_utime;     /* User time consumed */
    clock_t  si_stime;     /* System time consumed */
    sigval_t si_value;     /* Signal value */
    int      si_int;       /* POSIX.1b signal */
    void     *si_ptr;       /* POSIX.1b signal */
    int      si_overrun;   /* Timer overrun count;
                           POSIX.1b timers */
    int      si_timerid;   /* Timer ID; POSIX.1b timers */
    void     *si_addr;     /* Memory location which caused fault */
    long     si_band;      /* Band event (was int in
                           glibc 2.3.2 and earlier) */
    int      si_fd;        /* File descriptor */
    short    si_addr_lsb;  /* Least significant bit of address
                           (since Linux 2.6.32) */
    void     *si_lower;    /* Lower bound when address violation
                           occurred (since Linux 3.19) */
    void     *si_upper;    /* Upper bound when address violation
                           occurred (since Linux 3.19) */
    int      si_pkey;      /* Protection key on PTE that caused
                           fault (since Linux 4.6) */
    void     *si_call_addr; /* Address of system call instruction
                           (since Linux 3.5) */
    int      si_syscall;   /* Number of attempted system call
                           (since Linux 3.5) */
    unsigned int si_arch;  /* Architecture of attempted system call
                           (since Linux 3.5) */
}
```

信号相关函数_sigaction

```
void (*sa sigaction)(int, siginfo_t *, void *)
```

参数void *, 可以用来传递进程的上下文信息，系统提供了一个结构，ucontext_t作为参考

信号相关函数_sigsetjmp/siglongjmp

1. 函数的需求

在一个信号处理程序中，激活信号本身会被加入到信号屏蔽字中，如果longjmp从一个信号处理程序中跳出，那么信号屏蔽字怎么恢复，这一点原始的函数中并没有明确的要求。

2. int sigsetjmp(sigjmp buf env, int savemask)

如果savemask非0，则在env中将保存当前的信号屏蔽字

void siglongjmp(sigjmp buf env, int val)

如果env是由一个设置了非0的savemask的生成的，那么就会恢复当时的信号屏蔽字

信号相关函数_sigsuspend

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

```
    //返回-1， 并将errno设置位EINTR
```

将进程的信号屏蔽字设置为sigmask，
在捕捉到一个信号之前挂起，

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该函数之前的值。

信号相关函数_abort

void abort(void);

//将SIGABRT解除阻塞，并且给调用进程发送一个
//SIGABRT，随后终止进程

```
void myabort(void)
{
    sigset_t mask;
    struct sigaction action;
    /*信号的处理方式不可以设置为忽略，如果是，就设置为默认动作*/
    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);

    /*该信号不可以被阻塞*/
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT);
    sigprocmask(SIG_SETMASK, &mask, NULL);
    kill(getpid(), SIGABRT); //raise(SIGABRT)
    //如果设置了捕获函数，这里将会被捕获函数捕捉，如果没有设置，则会执行默认动作，终止进程

    fflush(NULL); //可以到达这里，说明设置了捕捉函数，且捕捉函数没有退出进程
    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);
    sigprocmask(SIG_SETMASK, &mask, NULL); //just in case
    kill(getpid(), SIGABRT);
    exit(1); //this should never be executed
}
```

system函数设计

```
int system(const char *command);
```

DESCRIPTION

The **system()** library function uses **fork(2)** to create a child process that executes the shell command specified in command using **execl(3)** as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) 0);
```

system() returns after the command has been completed.

During execution of the command, **SIGCHLD** will be blocked, and **SIGINT** and **SIGQUIT** will be ignored, in the process that calls **system()** (these signals will be handled according to their defaults inside the child process that executes command).

If command is **NULL**, then **system()** returns a status indicating whether a shell is available on the system.

1. 为什么要阻塞SIGCHLD，忽略SIGINT和SIGQUIT？

2. 如何去阻塞

3. 当一个SIGCHLD未决期间，wait和waitpid 返回了子进程的状态，那么在解除阻塞后，该信号还会不会抵达？
标准规定不允许抵达，但是Linux没有实现。

函数sleep相关及其设计

```
unsigned int sleep(unsigned int seconds)  
    //返回0或者未休眠完的秒  
    //在达到指定的定时时间，或者捕捉到一个信号，并从信号处理程序返回
```

```
int nanosleep(const struct timespec *reqtp, struct timespec *remtp)  
    //休眠到要求的时间，返回0，出错，返回-1  
    //函数挂起调用进程，直到超时或者某个信号中断， 如果某个信号中断，那么remtp将会被设置为剩余的定时时间
```

```
int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *reqtp, struct timespec *remtp);  
    //休眠到要求时间，返回0，出错，返回错误码  
    //clock_id 表明休眠的计数时钟类型  
    //flags表明休眠的时间相对性，0代表相对时间[休眠长度]，TIMER_ABSTIME代表绝对时间[休眠到这个时间点]
```

信号相关函数_sigqueue

```
int sigqueue(pid_t pid, int signo, const union sigval value);  
//成功，返回0， 出错，返回-1
```

在linux下，排队只对SIGRTMIN-SIGRTMAX之间的信号起作用， 针对于该范围外的信号，只起到信号传参的作用

信号——作业控制信号

SIGCHLD：子进程已经停止或者终止

SIGCONT：如果进程已经终止，则让其继续运行

SIGSTOP：停止信号

SIGTSTP：交互式停止

SIGTTIN：后台进程组试图读终端

SIGTTOU：后台进程组视图写终端

1. 当对一个进程产生四种停止信号的任意一种时， 对该进程的SIGCONT未决信号就被丢弃， 当对一个进程产生SIGCONT， 对该进程的任一未决停止信号被丢弃

2. 当对一个停止的信号产生一个SIGCONT信号的时， 该进程就继续， 即使该信号是被阻塞或忽略， 说明， 如果进程设置了阻塞SIGCONT， 这样如果收到一个SIGCONT， 会表现在未决信号集中，但是信号还是会导致停止的信号继续工作。

信号相关函数_信号编号与信号名映射

`const char *const *_sys_siglist;` //位于signal.h, 是一个存储对应信号名称的字符串指针数组

`void psignal(int signo, const char *msg);`
类似于perror, 以 “msg: 信号说明\n”的格式输出到标准错误流

`void psiginfo(const siginfo_t *info, const char *msg);` //输出到标准错误流, 详细信息

`char *strsignal(int signo);` //返回指向该信号的字符串的指针