Signal高级特性

更灵活的信号处置: sigaction

*原型: int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);*

```
struct sigaction {
    union {
        void (*sa_handler)(int);
        void (*sa_handler)(int, siginfo_t *, void *);
    } __sigaction_handler;          /* 二选一的信号处理函数 */

    sigset_t sa_mask;               /* 信号掩码，在调用信号处理程序之前，将改组信号中当前
                                       未处于进程掩码之列的任何信号自动添加到进程掩码中 */

    int      sa_flags;              /* 位掩码字段，用于控制信号处理过程的各种选项 */

    void     (*sa_restorer)(void);  /* 内部使用 */
};
```

act与oldact所指向的结构体

sigaction结构体

- sa_handler: 可有两个选择，一个是带有信号详细信息的，一个是不带有的
- sa_mask: 用户所定义的信号掩码，必须由sigemptyset()或者是sigfillset()初始化
- sa_flags
  - SA_NOCLDSTOP: 若sig为SIGCHLD信号，则子进程恢复运行或停止时，将不会产生此信号
  - SA_NOCLDWAIT: 若sig为SIGCHLD信号，则当子进程终止时不会将其转化为僵尸进程
  - SA_RESTART: 自动重启由信号处理程序所中断的系统调用
  - SA_SIGINFO: 调用信号处理程序时将携带额外参数，其中提供了关于信号的深入信息

```
struct sigaction sa;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;

if (sigaction(SIGINT, &sa, NULL) == -1) {
    perror("sigaction SIGINT");
}
```

使用sigaction()调用要比signal()调用更为麻烦一些，但是获得了更精细的控制，以及更多的特性

可重入与不可重入函数

在信号处理函数中，应尽量避免使用诸如stdio库等非可冲入函数: 它们是非线程安全的

stdio库在函数内部使用静态数据结构用于内部记账，库函数并不会为用于提供锁变量等线程保护机制，那么在多线程环境下，就会出现数据不一致的情况

系统调用的中断和重启

场景

1. 为某一个信号创建处理器函数
2. 进程发起一个阻塞的系统调用，如从socket中读取数据，进程阻塞，CPU转而执行其它任务
3. 之前创建了信号处理函数的信号由内核传递了过来，并且进程得到调度资源，转而执行信号处理函数

在如上场景下，当信号处理函数返回时会发生什么？阻塞的系统调用被中断，是否会自行重启？

默认情况下，系统调用将会失败，并且将errno设置为EINTR

这是个很有用的特性，可以据此并使用定时器来实现超时读取功能

但是，在更多的时候我们还是希望系统调用能自动重启，继续运行

在调用sigaction()添加信号处理函数时，指定SA_RESTART该标识位，内核将自动地重启被打断的系统调用

然而，有些系统调用即使指定了SA_RESTART，也不会自动重启

- select()、poll()、epoll()等I/O多路复用调用
- Linux所特有的epoll_wait()以及epoll_pwait()调用
- 对inotify文件描述符发起的read()调用
- 用于将进程挂起指定时间的系统调用以及库函数调用，如sleep()、nanosleep()和clock_nanosleep()