第六课　信号处理

一、基本概念

1. 中断

中止(注意不是终止)当前正在执行的程序，
转而执行其它任务。

硬件中断：来自硬件设备的中断。
软件中断：来自其它程序的中断。

2. 信号是一种软件中断

信号提供了一种以异步方式执行任务的机制。

3. 常见信号

SIGHUP(1)：连接断开信号
如果终端驱动检测到一个连接断开，
则将此信号发送给与该终端相关的控制进程(会话首进程)。
默认动作：终止。

SIGINT(2)：终端中断符信号
用户按中断键(Ctrl+C)，产生此信号，
并送至前台进程组的所有进程。
默认动作：终止。

SIGQUIT(3)：终端退出符信号
用户按退出键(Ctrl+\)，产生此信号，
并送至前台进程组的所有进程。
默认动作：终止+core。

SIGILL(4)：非法硬件指令信号
进程执行了一条非法硬件指令。
默认动作：终止+core。

SIGTRAP(5)：硬件故障信号
指示一个实现定义的硬件故障。常用于调试。
默认动作：终止+core。

SIGABRT(6)：异常终止信号
调用abort函数，产生此信号。
默认动作：终止+core。

SIGBUS(7)：总线错误信号
指示一个实现定义的硬件故障。常用于内存故障。
默认动作：终止+core。

SIGFPE(8)：算术异常信号

表示一个算术运算异常，例如除以0、浮点溢出等。
默认动作：终止+core。

SIGKILL(9)：终止信号
不能被捕获或忽略。常用于杀死进程。
默认动作：终止。

SIGUSR1(10)：用户定义信号
用户定义信号，用于应用程序。
默认动作：终止。

SIGSEGV(11)：段错误信号
试图访问未分配的内存，或向没有写权限的内存写入数据。
默认动作：终止+core。

SIGUSR2(12)：用户定义信号
用户定义信号，用于应用程序。
默认动作：终止。

SIGPIPE(13)：管道异常信号
写管道时读进程已终止，
或写SOCK_STREAM类型套接字时连接已断开，均产生此信号。
默认动作：终止。

SIGALRM(14)：闹钟信号
以alarm函数设置的计时器到期，
或以setitimer函数设置的真实计时器到期，均产生此信号。
默认动作：终止。

SIGTERM(15)：终止信号
由kill命令发送的系统默认终止信号。
默认动作：终止。

SIGSTKFLT(16)：数协器栈故障信号
表示数学协处理器发生栈故障。
默认动作：终止。

SIGCHLD(17)：子进程状态改变信号
在一个进程终止或停止时，将此信号发送给其父进程。
默认动作：忽略。

SIGCONT(18)：使停止的进程继续
向处于停止状态的进程发送此信号，令其继续运行。
默认动作：继续/忽略。

SIGSTOP(19)：停止信号
不能被捕获或忽略。停止一个进程。
默认动作：停止。

SIGTSTP(20)：终端停止符信号。
用户按停止键(Ctrl+Z)，产生此信号，
并送至前台进程组的所有进程。
默认动作：停止。

SIGTTIN(21)：后台读控制终端信号
后台进程组中的进程试图读其控制终端，产生此信号。

默认动作：停止。

SIGTTOU(22)：后台写控制终端信号
后台进程组中的进程试图写其控制终端，产生此信号。
默认动作：停止。

SIGURG(23)：紧急情况信号
有紧急情况发生，或从网络上接收到带外数据，产生此信号。
默认动作：忽略。

SIGXCPU(24)：超过CPU限制信号
进程超过了其软CPU时间限制，产生此信号。
默认动作：终止+core。

SIGXFSZ(25)：超过文件长度限制信号
进程超过了其软文件长度限制，产生此信号。
默认动作：终止+core。

SIGVTALRM(26)：虚拟闹钟信号
以setitimer函数设置的虚拟计时器到期，产生此信号。
默认动作：终止。

SIGPROF(27)：实用闹钟信号
以setitimer函数设置的实用计时器到期，产生此信号。
默认动作：终止。

SIGWINCH(28)：终端窗口大小改变信号
以ioctl函数更改窗口大小，产生此信号。
默认动作：忽略。

SIGIO(29)：异步I/O信号
指示一个异步I/O事件。
默认动作：终止。

SIGPWR(30)：电源失效信号
电源失效，产生此信号。
默认动作：终止。

SIGSYS(31)：非法系统调用异常。
指示一个无效的系统调用。
默认动作：终止+core。

4. 不可靠信号(非实时信号)

1) 那些建立在早期机制上的信号被称为“不可靠信号”。
   小于SIGRTMIN(34)的信号都是不可靠信号。

2) 不支持排队，可能会丢失。同一个信号产生多次，
   进程可能只收到一次该信号。

3) 进程每次处理完这些信号后，
   对相应信号的响应被自动恢复为默认动作，
   除非显示地通过signal函数重新设置一次信号处理程序。

5. 可靠信号(实时信号)

~~~~~~~~~~~~~~~~~~~~

1) 位于[SIGRTMIN(34),
   SIGRTMAX(64)]区间的信号都是可靠信号。

2) 支持排队，不会丢失。

3) 无论可靠信号还是不可靠信号，
   都可以通过sigqueue/sigaction函数发送/安装，
   以获得比其早期版本kill/signal函数更可靠的使用效果。

6. 信号的来源
~~~~~~~~~~

1) 硬件异常：除0、无效内存访问等。
   这些异常通常被硬件(驱动)检测到，并通知系统内核。
   系统内核再向引发这些异常的进程递送相应的信号。

2) 软件异常：通过
   kill/raise/alarm/setitimer/sigqueue
   函数产生的信号。

7. 信号处理
~~~~~~~~~~

1) 忽略。

2) 终止进程。

3) 终止进程同时产生core文件。

4) 捕获并处理。当信号发生时，
   内核会调用一个事先注册好的用户函数(信号处理函数)。

范例：loop.c

a.out
按中断键(Ctrl+C)，发送SIGINT(2)终端中断符信号。

a.out
按退出键(Ctrl+\)，发送SIGQUIT(3)终端退出符信号。

二、signal

```
#include <signal.h>

typedef void (*sighandler_t) (int);

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

signum - 信号码，也可使用系统预定义的常量宏，
          如SIGINT等。

handler - 信号处理函数指针或以下常量：
~~~~~~~~~~

SIG_IGN: 忽略该信号;
SIG_DFL: 默认处理。

成功返回原来的信号处理函数指针或SIG_IGN/SIG_DFL常量,
失败返回SIG_ERR。

1. 在某些Unix系统上,
通过signal函数注册的信号处理函数只一次有效,
即内核每次调用信号处理函数前,
会将对该信号的处理自动恢复为默认方式。
为了获得持久有效的信号处理,
可以在信号处理函数中再次调用signal函数,
重新注册一次。

例如:

```
void sigint (int signum) {
    ...
    signal (SIGINT, sigint);
}

int main (void) {
    ...
    signal (SIGINT, sigint);
    ...
}
```

2. SIGKILL/SIGSTOP信号不能被忽略, 也不能被捕获。

3. 普通用户只能给自己的进程发送信号,
root用户可以给任何进程发送信号。

范例: signal.c

三、子进程的信号处理

1. 子进程会继承父进程的信号处理方式,
直到子进程调用exec函数。

范例: fork.c

2. 子进程调用exec函数后,
exec函数将被父进程设置为捕获的信号恢复至默认处理,
其余保持不变。

范例: exec.c

四、发送信号

1. 键盘

Ctrl+C - SIGINT(2), 终端中断

Ctrl+\ - SIGQUIT(3), 终端退出
Ctrl+Z - SIGTSTP(20), 终端暂停

2. 错误

除0   - SIGFPE(8), 算术异常
非法内存访问 - SIGSEGV(11), 段错误
硬件故障   - SIGBUS(7), 总线错误

3. 命令

kill -信号 进程号

4. 函数

1) kill

#include <signal.h>

int kill (pid_t pid, int sig);

成功返回0，失败返回-1。

pid > 0 - 向pid进程发送sig信号。

pid = 0 - 向同进程组的所有进程发送sig信号。

pid = -1 - 向所有进程发送sig信号，
           前提是调用进程有向其发送信号的权限。

pid < -1 - 向绝对值等于pid的进程组中的所有进程发送sig信号。

0信号为空信号。
若sig取0，则kill函数仍会执行错误检查，
但并不实际发送信号。这常被用来确定一个进程是否存在。
向一个不存在的进程发送信号，会返回-1，且errno为ESRCH。

范例：kill.c

2) raise

#include <signal.h>

int raise (int sig);

向调用进程自身发送sig信号。成功返回0，失败返回-1。

范例：raise.c

五、pause

#include <unistd.h>

```
int pause (void);
```

1. 使调用进程进入睡眠状态，
   直到有信号终止该进程或被捕获。

2. 只有调用了信号处理函数并从中返回以后，
   该函数才会返回。

3. 该函数要么不返回(未捕获到信号)，
   要么返回-1(被信号中断)，
   errno为EINTR。

4. 相当于没有时间限制的sleep函数。

范例：pause.c

六、sleep

```
#include <unistd.h>

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

1. 使调用进程睡眠seconds秒，
   除非有信号终止该进程或被捕获。

2. 只有睡够seconds秒，
   或调用了信号处理函数并从中返回以后，
   该函数才会返回。

3. 该函数要么返回0(睡够)，
   要么返回剩余秒数(被信号中断)。

4. 相当于有时间限制的pause函数。

范例：sleep.c

```
#include <unistd.h>

int usleep (useconds_t usec);
```

使调用进程睡眠usec微秒，
除非有信号终止该进程或被捕获。
成功返回0，失败返回-1。

七、alarm

```
#include <unistd.h>

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

1. 使内核在seconds秒之后，
   向调用进程发送SIGALRM(14)闹钟信号。

范例：clock.c

2. SIGALRM信号的默认处理是终止进程。

3. 若之前已设过定时且尚未超时，
   则调用该函数会重新设置定时，
   并返回之前定时的剩余时间。

4. seconds取0表示取消之前设过且尚未超时的定时。

范例：alarm.c

八、信号集与信号阻塞(信号屏蔽)

1. 信号集

1) 多个信号的集合类型：
   sigset_t，128个二进制位，每个位代表一个信号。

2) 相关函数

```
#include <signal.h>

// 将信号集set中的全部信号位置1
int sigfillset (sigset_t* set);

// 将信号集set中的全部信号位清0
int sigemptyset (sigset_t* set);

// 将信号集set中与signum对应的位置1
int sigaddset (sigset_t* set, int signum);

// 将信号集set中与signum对应的位清0
int sigdelset (sigset_t* set, int signum);
```

成功返回0，失败返回-1。

```
// 判断信号集set中与signum对应的位是否为1
int sigismember (const sigset_t* set, int signum);
```

若信号集set中与signum对应的位为1，则返回1，否则返回0。

范例：sigset.c

2. 信号屏蔽

1) 当信号产生时，系统内核会在其所维护的进程表中，
   为特定的进程设置一个与该信号相对应的标志位，
   这个过程称为递送(delivery)。

2) 信号从产生到完成递送之间存在一定的时间间隔。
   处于这段时间间隔中的信号状态，称为未决(pending)。

3) 每个进程都有一个信号掩码(signal mask)。
   它实际上是一个信号集，
   其中包括了所有需要被屏蔽的信号。

4) 可以通过sigprocmask函数，
   检测和修改调用进程的信号掩码。
   也可以通过sigpending函数，
   获取调用进程当前处于未决状态的信号集。

5) 当进程执行诸如更新数据库等敏感任务时，
   可能不希望被某些信号中断。
   这时可以暂时屏蔽(注意不是忽略)这些信号，
   使其滞留在未决状态。
   待任务完成以后，再回过头来处理这些信号。

6) 在信号处理函数的执行过程中，
   这个正在被处理的信号总是处于信号掩码中。

```
#include <signal.h>

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

成功返回0，失败返回-1。

how　- 修改信号掩码的方式，可取以下值：

    SIG_BLOCK: 新掩码是当前掩码和set的并集
    (将set加入信号掩码)；

    SIG_UNBLOCK: 新掩码是当前掩码和set补集的交集
    (从信号掩码中删除set)；

    SIG_SETMASK: 新掩码即set(将信号掩码设为set)。

set　- NULL则忽略。

oldset - 备份以前的信号掩码，NULL则不备份。

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

set - 输出，调用进程当前处于未决状态的信号集。

成功返回0，失败返回-1。

注意：对于不可靠信号，
通过sigprocmask函数设置信号掩码以后，
相同的被屏蔽信号只会屏蔽第一个，
并在恢复信号掩码后被递送，其余的则直接忽略掉。
而对于可靠信号，
则会在信号屏蔽时按其产生的先后顺序排队，
一旦恢复信号掩码，这些信号会依次被信号处理函数处理。

范例：sigmask.c

九、sigaction

```
#include <signal.h>

int sigaction (
    int                     signum, // 信号码
    const struct sigaction* act,    // 信号处理方式
    struct sigaction*       oldact  // 原信号处理方式
                                    // (可为NULL)
);

struct sigaction {
    void     (*sa_handler) (int);
                                    // 信号处理函数指针1
    void     (*sa_sigaction) (int, siginfo_t*, void*);
                                    // 信号处理函数指针2
    sigset_t sa_mask;               // 信号掩码
    int      sa_flags;              // 信号处理标志
    void     (*sa_restorer)(void);
                                    // 保留，NULL
};
```

成功返回0，失败返回-1。

1. 缺省情况下，在信号处理函数的执行过程中，
   会自动屏蔽这个正在被处理的信号，
   而对于其它信号则不会屏蔽。
   通过sigaction::sa_mask成员可以人为指定，
   在信号处理函数的执行过程中，
   需要加入进程信号掩码中的信号，
   并在信号处理函数执行完之后，
   自动解除对这些信号的屏蔽。

2. sigaction::sa_flags可为以下值的位或：

SA_ONESHOT/SA_RESETHAND - 执行完一次信号处理函数后，
即将对此信号的处理恢复为
默认方式(这也是老版本
signal函数的缺省行为)。

SA_NODEFER/SA_NOMASK - 在信号处理函数的执行过程中，
不屏蔽这个正在被处理的信号。

SA_NOCLDSTOP - 若signum参数取SIGCHLD，
则当子进程暂停时，
不执行信号处理函数。

SA_RESTART - 系统调用一旦被signum参数
所表示的信号中断，
会自行重启。

缺省情况下，在一个系统调用被阻塞期间，收到信号，
系统将中断这个被阻塞的系统调用，转而执行相应的信号处理函数。
待信号处理函数执行完以后，之前被中断的系统调用返回失败，
同时置errno为EINTR。为了继续之前的系统调用，
程序中往往需要特殊处理：

```
    ssize_t len;
    char buf[256];
again:
    if ((len = read (fd, buf, sizeof (buf))) == -1) {
        if (errno == EINTR)
            goto again;
        perror ("read");
        return -1;
    }
```

但如果使用了SA_RESTART标志，被信号中断的系统调用，
将在相应信号处理函数执行完以后自动恢复，
继续被中断前的操作。

```
    ssize_t len;
    char buf[256];
    if ((len = read (fd, buf, sizeof (buf))) == -1) {
        perror ("read");
        return -1;
    }
```

```
SA_SIGINFO              - 使用信号处理函数指针2，
                          通过该函数的第二个参数，
                          提供更多信息。
```

```
typedef struct siginfo {
    pid_t    si_pid;   // 发送信号的PID
    sigval_t si_value; // 信号附加值
                       // (需要配合sigqueue函数)
    ...
}   siginfo_t;

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

范例：sigact.c、restart.c

十、sigqueue

```
#include <signal.h>

int sigqueue (pid_t pid, int sig,
    const union sigval value);
```

向pid进程发送sig信号，附加value值(整数或指针)。
成功返回0，失败返回-1。

范例：sigque.c

注意：sigqueue函数对不可靠信号不做排队，会丢失信号。

十一、计时器

---

1. 系统为每个进程维护三个计时器

1) 真实计时器：
   程序运行的实际时间。

2) 虚拟计时器：
   程序运行在用户态所消耗的时间。

3) 实用计时器：
   程序运行在用户态和内核态所消耗的时间之和。

实际时间(真实计时器) = 用户时间(虚拟计时器) + 内核时间 + 等待时间(I/O、睡眠等)

                            (实用计时器)

2. 为进程设定计时器

1) 用指定的初始间隔和重复间隔为进程设定好计时器后，
   该计时器就会定时地向进程发送时钟信号。

2) 三个计时器所发送的时钟信号分别为：

SIGALRM - 真实计时器
SIGVTALRM - 虚拟计时器
SIGPROF - 实用计时器

3) 获取/设置计时器

```
#include <sys/time.h>

int getitimer (int which,
    struct itimerval* curr_value);
```

获取计时器设置。成功返回0，失败返回-1。

```
int setitimer (int which,
    const struct itimerval* new_value,
    struct itimerval* old_value);
```

设置计时器。成功返回0，失败返回-1。

which   - 指定哪个计时器，取值：

    ITIMER_REAL: 真实计时器；
    ITIMER_VIRTUAL: 虚拟计时器；
    ITIMER_PROF: 实用计时器。

curr_value - 当前设置。

new_value - 新的设置。

old_value - 旧的设置(可为NULL)。

```
struct itimerval {
    struct timeval it_interval;
    // 重复间隔(每两个时钟信号的时间间隔),
    // 取0将使计时器在发送第一个信号后停止
    struct timeval it_value;
    // 初始间隔(从调用setitimer函数到第一次发送
    // 时钟信号的时间间隔), 取0将立即停止计时器
};

struct timeval {
    long tv_sec;  // 秒数
    long tv_usec; // 微秒数
};
```

范例：timer.c