LINUX 信号处理

1. 信号概念

信号是进程在运行过程中，由自身产生或由进程外部发过来的消息（事件）。信号是硬件中断的软件模拟(软中断)。每个信号用一个整型常量宏表示，以 SIG 开头，比如 SIGCHLD、SIGINT 等，它们在系统头文件**<signal.h>**中定义,也可以通过在 shell 下键入 **kill –l** 查看信号列表，或者键入 **man 7 signal** 查看更详细的说明。

信号的生成来自内核，让内核生成信号的请求来自 3 个地方：

- 用户：用户能够通过输入 CTRL+c、Ctrl+\，或者是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号；
- 内核：当进程执行出错时，内核会给进程发送一个信号，例如非法段存取(内存访问违规)、浮点数溢出等；
- 进程：一个进程可以通过系统调用 kill 给另一个进程发送信号，一个进程可以通过信号和另外一个进程进行通信。

由进程的某个操作产生的信号称为同步信号(synchronous signals),例如除 0；由像用户击键这样的进程外部事件产生的信号叫做异步信号(asynchronous signals)。

进程接收到信号以后，可以有如下 3 种选择进行处理：

- 接收默认处理：接收默认处理的进程通常会导致进程本身消亡。例如连接到终端的进程，用户按下 CTRL+c，将导致内核向进程发送一个 SIGINT 的信号，进程如果不对该信号做特殊的处理，系统将采用默认的方式处理该信号，即终止进程的执行； signal(SIGINT,SIG_DFL);
- 忽略信号：进程可以通过代码，显示地忽略某个信号的处理，例如：signal(SIGINT,SIG_IGN);但是某些信号是不能被忽略的,例如 9 号信号；
- 捕捉信号并处理：进程可以事先注册信号处理函数，当接收到信号时，由信号处理函数自动捕捉并且处理信号。

有两个信号既不能被忽略也不能被捕捉，它们是 SIGKILL 和 SIGSTOP。即进程接收到这两个信号后，只能接受系统的默认处理，即终止进程。SIGSTOP 是暂停进程。

2. signal 信号处理机制

可以用函数 signal 注册一个信号捕捉函数。原型为：

```
#include <signal.h>
typedef void (*sighandler_t)(int);   //函数指针
sighandler_t    signal(int signum, sighandler_t handler);
```

signal 的第 1 个参数 signum 表示要捕捉的信号，第 2 个参数是个函数指针，表示要对该信号进行捕捉的函数，该参数也可以是 SIG_DFL(表示交由系统缺省处理，相当于白注册了)或 SIG_IGN(表示忽略掉该信号而不做任何处理)。signal 如果调用成功，返回以前该信号的处理函数的地址，否则返回 SIG_ERR。

sighandler_t 是信号捕捉函数，由 signal 函数注册，注册以后，在整个进程运行过程中均有效，并且对不同的信号可以注册同一个信号捕捉函数。该函数只有一个整型参数，表示信号值。

示例：

1、 捕捉终端 CTRL+c 产生的 SIGINT 信号：

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
void SignHandler(int iSignNo)
{
    printf("Capture sign no:%d\n",iSignNo);

}

int main()
{
    signal(SIGINT,SignHandler);
    while(1)
        sleep(1);
    return 0;
}
```

该程序运行起来以后，通过按 CTRL+c 将不再终止程序的运行（或者另开一个终端，然后发送消息：“kill –s 2 进程号”或者“kill -2 进程号”，可以实现 Ctrl + c 同样的效果。因为 CTRL+c 产生的 SIGINT 信号已经由进程中注册的 SignHandler 函数捕捉了。该程序可以通过 Ctrl+\终止，因为组合键 Ctrl+\能够产生 SIGQUIT 信号，而该信号的捕捉函数尚未在程序中注册。

2、 忽略掉终端 CTRL+c 产生的 SIGINT 信号：

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
    signal(SIGINT,SIG_IGN);
    while(1)
        sleep(1);
    return 0;
}
```

该程序运行起来以后，将 CTRL+C 产生的 SIGINT 信号忽略掉了，所以 CTRL＋C 将不再能使该进程终止，要终止该进程，可以向进程发送 SIGQUIT 信号，即组合键 CTRL+\。或者另外开一个端口，然后执行 ps –aux 查看进程，发现该进程号之后用 kill -9 进程号 杀掉该进程。

3、 接受信号的默认处理,接受默认处理就相当于没有写信号处理程序：

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

```
int main()
{
    signal(SIGINT,SIG_DFL);
    while(1)
        sleep(1);
    return 0;
}
```

3. sigaction 信号处理机制

3.1. 信号处理情况分析

在 signal 处理机制下，还有许多特殊情况需要考虑：

1、注册一个信号处理函数，并且处理完毕一个信号之后，是否需要重新注册，才能够捕捉下一个信号；（不需要）

2、如果信号处理函数正在处理信号，并且还没有处理完毕时，又发生了一个同类型的信号，这时该怎么处理；（挨着执行），**后续相同信号忽略（会多执行一次）。**

3、如果信号处理函数正在处理信号，并且还没有处理完毕时，又发生了一个不同类型的信号，这时该怎么处理；**（跳转去执行另一个信号，之后再执行剩下的没有处理完的信号）**

4、如果程序阻塞在一个系统调用(如 read(...))时，发生了一个信号，这时是让系统调用返回错误再接着进入信号处理函数，还是先跳转到信号处理函数，等信号处理完毕后，系统调用再返回。

相互打断就是一次，相同信号处理函数不重入

示例：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
int g_iSeq = 0;

void SignHandler(int iSignNo)
{
    int iSeq = g_iSeq++;
    printf("%d Enter SignHandler,signo:%d\n",iSeq,iSignNo);
    sleep(3);
    printf("%d Leave SignHandler,signo:%d\n",iSeq,iSignNo);
}

int main()
{
    char szBuf[8];
    int iRet;
    signal(SIGINT,SignHandler);       //不同的信号调用同一个处理函数
    signal(SIGQUIT,SignHandler);
```

```
        do{
            iRet = read(STDIN_FILENO,szBuf,sizeof(szBuf)-1);
            if(iRet < 0){

                perror("read fail.");
                break;
            }
            szBuf[iRet] = 0;
            printf("Get: %s",szBuf);
        }while(strcmp(szBuf,"quit\n") != 0);
        return 0;
    }
```

程序运行时，针对于如下几种输入情况(要输入得快)，看输出结果：

1、 [CTRL+c] [CTRL+c]　　（一个一个挨着执行）
2、 [CTRL+c] [CTRL+\]　　（先执行 c 的进入，被\打断，转而执行\，最后执行 c 的退出）
3、 hello [CTRL+\] [Enter]　　（先执行中断，没有任何输出）
4、 [CTRL+\] hello [Enter]　　（先执行中断，输出内容）
5、 hel [CTRL+\] lo[Enter]　　（先执行中断，只输出 lo）

3.2. sigaction 信号处理注册

函数原型：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

sigaction 也用于注册一个信号处理函数

参数 signum 为需要捕捉的信号

参数 act 是一个结构体，里面包含信号处理函数地址、处理方式等信息

参数 oldact 是一个传出参数，sigaction 函数调用成功后，oldact 里面包含以前对 signum 的处理方式的信息，通常为 NULL

如果函数调用成功，将返回 0，否则返回-1

结构体 struct sigaction(注意名称与函数 sigaction 相同)的原型为：

```
struct sigaction {
    void (*sa_handler)(int);        //老类型的信号处理函数指针
    void (*sa_sigaction)(int, siginfo_t *, void *);//新类型的信号处理函数指针
    sigset_t sa_mask;               //将要被阻塞的信号集合
    int sa_flags;                   //信号处理方式掩码 ( ➔ SA_SIGINFO )
    void (*sa_restorer)(void);      //保留，不要使用
};
```

该结构体的各字段含义及使用方式：

1、字段 sa_handler 是一个函数指针，用于指向原型为 void handler(int)的信号处理函数地址，即老类型　　的信号处理函数（如果用这个再将 sa_flags = 0,就等同于 signal()函数）

2、字段 sa_sigaction 也是一个函数指针，用于指向原型为：

```
void handler(int iSignNum, siginfo_t *pSignInfo, void *pReserved);
```

的信号处理函数，即新类型的信号处理函数

该函数的三个参数含义为：

iSignNum：传入的信号

pSignInfo：与该信号相关的一些信息，它是个结构体

pReserved：保留，现没用，通常为 NULL

3、字段 sa_handler 和 sa_sigaction 只应该有一个生效，如果想采用老的信号处理机制，就应该让 sa_handler 指向正确的信号处理函数，并且让字段 sa_flags 为 **0**；否则应该让 sa_sigaction 指向正确的信号处理函数，并且让字段 sa_flags 包含 **SA_SIGINFO** 选项

4、字段 sa_mask 是一个包含信号集合的结构体，**该结构体内的信号表示在进行信号处理时，将要被阻塞的信号**。针对 sigset_t 结构体，有一组专门的函数对它进行处理，它们是：

```
#include <signal.h>
int sigemptyset(sigset_t *set);                  //清空信号集合 set
int sigfillset(sigset_t *set);                   //将所有信号填充进 set 中
int sigaddset(sigset_t *set, int signum);        //往 set 中添加信号 signum
int sigdelset(sigset_t *set, int signum);        //从 set 中移除信号 signum
int sigismember(const sigset_t *set, int signum); //判断 signum 是否包含在 set 中(是:返回 1,否:0）
int sigpending(sigset_t *set);                   //将被阻塞的信号集合由参数 set 指针返回(挂起信号)
```

其中，对于函数 sigismember 而言，如果 signum 在 set 集中，则返回 1；不在，则返回 0；出错时返回-1.其他的函数都是成功返回 0，失败返回-1.

例如，如果打算在处理信号 SIGINT 时，只阻塞对 SIGQUIT 信号的处理，可以用如下方法：

```
struct sigaction act;
act.sa_flags = SA_SIGINFO;
act.sa_sigaction = newHandler;
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask, SIGQUIT);
sigaction(SIGINT,&act,NULL);
```

5、字段 sa_flags 是一组掩码的合成值，指示信号处理时所应该采取的一些行为，各掩码的含义为：

掩码	描述
SA_RESETHAND	处理完毕要捕捉的信号后，将自动撤消信号处理函数的注册，即必须再重新注册信号处理函数，才能继续处理接下来产生的信号。该选项不符合一般的信号处理流程，现已经被废弃。
SA_NODEFER	在处理信号时，如果又发生了其它的信号，则立即进入其它信号的处理，等其它信号处理完毕后，再继续处理当前的信号，即**递规**地处理。**（不常用）不断重入，次数不丢失**
SA_RESTART	如果在发生信号时，程序正阻塞在某个系统调用，例如调用 read()函数，则在处理完毕信号后，接着从阻塞的系统返回。如果不指定该参数，中断处理完毕之后，read 函数读取失败。
SA_SIGINFO	指示结构体的信号处理函数指针是哪个有效，如果 **sa_flags 包含该掩码，则 sa_sigaction 指针有效，否则是 sa_handler 指针有效**。（常用）

Example1：用 sigaction 实现和 signal（只能传递一个参数）一样的功能。

```
#include<signal.h>
#include<stdio.h>
void handle(int signo)
{
    printf("signo: %d\n",signo);
```

```
}
main()
{
    struct sigaction st;
    st.sa_handler = handle;
    st.sa_flags = 0;
    sigaction(SIGINT,&st,NULL);
    while(1)
    {
        sleep(1);
    }
}
```

Example2：用 sigaction 实现调用新的信号处理函数

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
int g_iSeq = 0;
void SignHandlerNew(int iSignNo,siginfo_t *pInfo,void *pReserved)
{
    int iSeq = g_iSeq++;
    printf("%d Enter SignHandlerNew,signo:%d\n",iSeq,iSignNo);
    sleep(3);
    printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
}
int main()
{
    struct sigaction act;
    act.sa_sigaction = SignHandlerNew;
    act.sa_flags = SA_SIGINFO;
    sigaction(SIGINT,&act,NULL);
    sigaction(SIGQUIT,&act,NULL);
    while(1)
    {
        sleep(1);
    }
    return 0;
}
```

练习与验证：

针对于先前的 5 种输入情况，给下面代码再添加一些代码，使之能够进行如下各种形式的响应：

1、[CTRL+c] [CTRL+c]时，第 1 个信号处理阻塞第 2 个信号处理；

2、[CTRL+c] [CTRL+c]时，第 1 个信号处理时，允许递规地第 2 个信号处理；

3、[CTRL+c] [CTRL+\]时，第 1 个信号阻塞第 2 个信号处理；

4、read 不要因为信号处理而返回失败结果。

```
#include <stdio.h>
```

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

int g_iSeq = 0;

void SignHandlerNew(int iSignNo,siginfo_t *pInfo,void *pReserved)
{
    int iSeq = g_iSeq++;
    printf("%d Enter SignHandlerNew,signo:%d.\n",iSeq,iSignNo);
    sleep(3);
    printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
}

int main()
{
    char szBuf[8] = {0};
    int iRet = 0;
    struct sigaction act;
    act.sa_sigaction = SignHandlerNew;
    act.sa_flags = SA_SIGINFO | SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, SIGQUIT);
    sigaction(SIGINT,&act,NULL);
    do{
        iRet = read(STDIN_FILENO,szBuf,sizeof(szBuf)-1);
        if(iRet < 0){
            perror("read fail");
            break;
        }
        szBuf[iRet] = 0;
        printf("Get: %s",szBuf);
    }while(strcmp(szBuf,"quit\n") != 0);
    return 0;
}
```

你会发现，当一个信号被阻塞之后，在解除阻塞之前，该信号发生了多次，但是解除阻塞的时候，内核只会向进程发送一个信号而已，而不管在其阻塞期间有多少个信号产生，因为 linux 并不会对信号进行排队。另外，这里用到了打断 read 输入的中断处理，必须要加参数 SA_RESTART，对于 signal 函数而言，它安装的信号处理函数，系统默认会自动重启被中断的系统调用，而不是让它出错返回。而对于 sigaction 函数而言，必须要自己指定 SA_RESTART 实现重启功能，如果不指定则会 read 失败，提示 read 的时候中断发生。

3.3. sigprocmask 信号阻塞

函数 sigaction 中设置的被阻塞信号集合只是针对于要处理的信号，例如

```
struct sigaction act;
```

```
        sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask,SIGQUIT);
        sigaction(SIGINT,&act,NULL);
```

表示只有在处理信号 SIGINT 时，才阻塞信号 SIGQUIT；**(重点区分)**

函数 sigprocmask 是全程阻塞，在 sigprocmask 中设置了阻塞集合后，被阻塞的信号将不能再被信号处理函数捕捉，直到重新设置阻塞信号集合。

原型为：

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

参数 how 的值为如下 3 者之一：

a：SIG_BLOCK ,将参数 2 的信号集合添加到进程原有的阻塞信号集合中

b：SIG_UNBLOCK ,从进程原有的阻塞信号集合移除参数 2 中包含的信号

c：SIG_SETMASK，重新设置进程的阻塞信号集为参数 2 的信号集

参数 set 为阻塞信号集

参数 oldset 是传出参数，存放进程原有的信号集，通常为 NULL

示例：添加全程阻塞

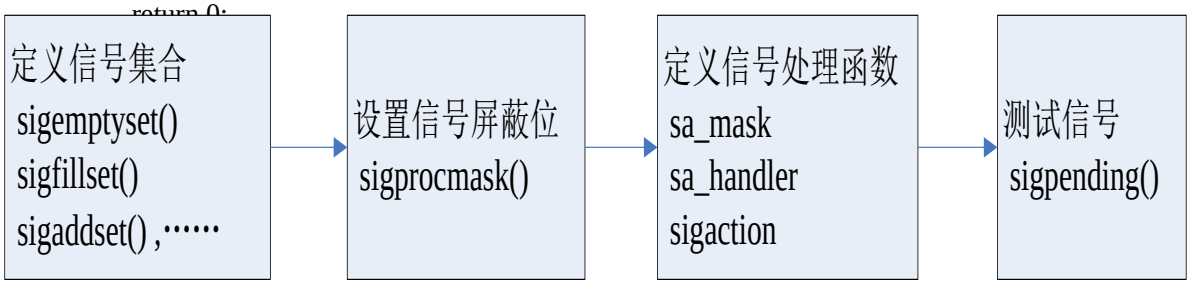
```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

int g_iSeq=0;

void SignHandlerNew(int iSignNo,siginfo_t *pInfo,void *pReserved)
{
    int iSeq=g_iSeq++;
    printf("%d Enter SignHandlerNew,signo:%d\n",iSeq,iSignNo);
    sleep(3);
    printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
}

int main()
{
    char szBuf[8];
    int iRet;

    //屏蔽掉 SIGQUIT 信号
    sigset_t sigSet;
    sigemptyset(&sigSet);
    sigaddset(&sigSet,SIGQUIT);
    sigprocmask(SIG_BLOCK,&sigSet,NULL);

        struct sigaction act;
    act.sa_sigaction=SignHandlerNew;
    act.sa_flags=SA_SIGINFO;
    sigemptyset(&act.sa_mask);
```

```
        sigaction(SIGINT,&act,NULL);

        while(1);
        return 0;
    }
```

实例：取消指定信号的全程阻塞。

```
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include <signal.h>
    int g_iSeq=0;
    void SignHandlerNew(int iSignNo,siginfo_t *pInfo,void *pReserved)
    {
        int iSeq=g_iSeq++;
        printf("%d Enter SignHandlerNew,signo:%d\n",iSeq,iSignNo);
        sleep(3);
        printf("%d Leave SignHandlerNew,signo:%d\n",iSeq,iSignNo);
    }

    int main()
    {
        //屏蔽掉 SIGINT 和 SIGQUIT 信号，SigHandlerNew 将不能再捕捉 SIGINT 和 SIGQUIT
        sigset_t sigSet;
        sigemptyset(&sigSet);
        sigaddset(&sigSet,SIGINT);
        sigaddset(&sigSet,SIGQUIT);
        sigprocmask(SIG_BLOCK,&sigSet,NULL);//将 SIGINT、SIGQUIT 屏蔽

        struct sigaction act;
        act.sa_sigaction=SignHandlerNew;
        act.sa_flags=SA_SIGINFO;
        sigemptyset(&act.sa_mask);
        sigaction(SIGINT,&act,NULL);
        sigaction(SIGQUIT,&act,NULL);
        int iCount = 0;
        while(1)
        {
            if(iCount > 3)
            {
                sigset_t sigSet2;
                sigemptyset(&sigSet2);
                sigaddset(&sigSet2, SIGINT);
                sigprocmask(SIG_UNBLOCK, &sigSet2, NULL);
            }
            iCount ++;
            sleep(2);
```

```
    }
    return 0;
```



Example3：利用 sigpending 测试信号，并采用上图的模型

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

void handler(int signum, siginfo_t* pInfo, void* pReversed)
{
    printf("receive signal %d \n",signum);
}

int main()
{
    sigset_t new_mask, old_mask, pending_mask;
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGINT);
    if(sigprocmask(SIG_BLOCK, &new_mask, &old_mask))
        printf("block signal SIGINT error\n");

    struct sigaction act;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = handler;
    if(sigaction(SIGINT, &act, NULL))
        printf("install signal SIGINT error\n");
    sleep(10);

    printf("now begin to get pending mask and unblock SIGINT\n");
    if(sigpending(&pending_mask) < 0)//将阻塞信号全部添加到 pending_mask 信号集中并返回
        printf("get pending mask error\n");

    if(sigismember(&pending_mask, SIGINT))
        printf("signal SIGINT is pending\n");

    if(sigprocmask(SIG_SETMASK, &old_mask, NULL) < 0)
```

```
            printf("unblock signal error\n");
        printf("signal unblocked\n");
        sleep(10);
        return 0;
    }
```

4. 用程序发送信号

4.1. kill 信号发送函数

原型为：

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

参数 pid 为将要接受信号的进程的 pid，可以通过 getpid()函数获得来给自身发送信号，还可以发送信号给指定的进程，此时 pid 有如下描述：

pid > 0　将信号发给 ID 为 pid 的进程
pid == 0　将信号发送给与发送进程属于同一个进程组的所有进程
pid < 0　将信号发送给进程组 ID 等于 pid 绝对值的所有进程
pid == -1 将信号发送给该进程有权限发送的系统里的所有进程

参数 sig 为要发送的信号

如果成功，返回 0，否则为-1

示例，输入结束后，将通过发送信号 SIGQUIT 把自己杀掉：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
        while(1){
        if(getchar()==EOF)      //运行之后输入没有反应，当按下 Ctrl+d（EOF），进程关闭
        kill(getpid(),SIGQUIT);
        }
    return 0;
}
```

除此之外，还可以向指定的进程发送信号：

程序 1：

```
#include <stdio.h>
int main()
{
    while(1);
    return 0;
}
```

程序 2：

```
# include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char* argv[])
{
    int pid = atoi(argv[1]);
        kill(pid,SIGQUIT);
    return 0;
}
```

首先运行程序 1，然后用 ps –aux 查看其进程号，假设位 11002。再运行程序 2　./2　11002 即可

5. 计时器与信号

5.1. 睡眠函数

Linux 下有两个睡眠函数，原型为：

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
void usleep(unsigned long usec);
```

函数 sleep 让进程睡眠 seconds 秒，函数 usleep 让进程睡眠 usec 微秒。

sleep 睡眠函数内部是用信号机制进行处理的，用到的函数有：

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
//告知自身进程，要进程在 seconds 秒后自动产生一个 SIGALRM 的信号
int pause(void); //将自身进程挂起，直到有信号发生时才从 pause 返回
```

示例：模拟睡眠 3 秒：

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

void SignHandler(int iSignNo)
{
    printf("signal:%d\n",iSignNo);
}

int main()
{
    signal(SIGALRM,SignHandler);
    alarm(3);       //等待 3 秒之后自动产生 SIGALRM 信号
    printf("Before pause().\n");
    pause();        //将进程挂起，直到有信号发生才退出挂起状态
```

```
        printf("After pause().\n");
        return 0;
    }
```

注意：因为 sleep 在内部是用 alarm 实现的，所以在程序中最好不要 sleep 与 alarm 混用，以免造成混乱。

5.2. 时钟处理

Linux 为每个进程维护 3 个计时器，分别是真实计时器、虚拟计时器和实用计时器。

- 真实计时器计算的是程序运行的实际时间；---直接
- 虚拟计时器计算的是程序运行在用户态时所消耗的时间(可认为是实际时间减掉(系统调用和程序睡眠所消耗)的时间)；---需要了解内核
- 实用计时器计算的是程序处于用户态和处于内核态所消耗的时间之和。---常用

例如：有一程序运行，在用户态运行了 5 秒，在内核态运行了 6 秒，还睡眠了 7 秒，则真实计算器计算的结果是 18 秒，虚拟计时器计算的是 5 秒，实用计时器计算的是 11 秒。

用指定的初始间隔和重复间隔时间为进程设定好一个计时器后，该计时器就会定时地向进程发送时钟信号。3 个计时器发送的时钟信号分别为：SIGALRM,SIGVTALRM 和 SIGPROF。

用到的函数与数据结构：

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

1. int getitimer(int which, struct itimerval *value);　　//获取计时器的设置

参数 which 指定哪个计时器，可选项为 ITIMER_REAL(真实计时器)、ITIMER_VIRTUAL(虚拟计时器、ITIMER_PROF(实用计时器))

参数 value 为一结构体的**传出参数**，用于传出该计时器的初始间隔时间和重复间隔时间

返回值：如果成功，返回 0，否则-1

2. int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue); //设置计时器

参数 which 指定哪个计时器，可选项为 ITIMER_REAL(真实计时器)、ITIMER_VIRTUAL(虚拟计时器、ITIMER_PROF(实用计时器))

参数 value 为一结构体的**传入参数**，指定该计时器的初始间隔时间和重复间隔时间

参数 ovalue 为一结构体传出参数，用于传出以前的计时器时间设置。

返回值：如果成功，返回 0，否则-1

```
struct itimerval {
    struct timeval it_interval; /* next value */          //重复间隔
    struct timeval it_value;    /* current value */       //初始间隔
};
struct timeval {
    long tv_sec;                /* seconds */                 //时间的秒数部分
    long tv_usec;               /* microseconds */    //时间的微秒部分
};
```

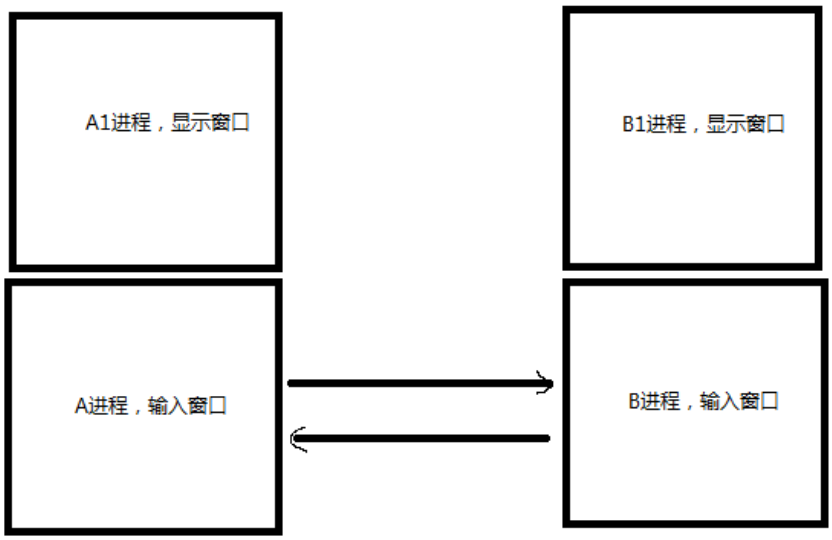
示例：启用真实计时器的进行时钟处理（获得当前系统时间，并一秒更新一次）

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

```
void sigHandler(int iSigNum)
{
    time_t tt;
    time(&tt);
    struct tm *pTm = gmtime(&tt);
    printf("%04d-%02d-%02d    %02d:%02d:%02d\n", (1900+pTm->tm_year), (1+pTm->tm_mon),
pTm->tm_mday, (8+pTm->tm_hour), pTm->tm_min, pTm->tm_sec);
}

void InitTime(int tv_sec, int tv_usec)
{
    signal(SIGALRM, sigHandler);
    alarm(0);
    struct itimerval tm;
    tm.it_value.tv_sec = tv_sec;
    tm.it_value.tv_usec = tv_usec;

    tm.it_interval.tv_sec = tv_sec;
    tm.it_interval.tv_usec = tv_usec;

    if(setitimer(ITIMER_REAL, &tm, NULL) == -1)
    {
        perror("setitimer error");
        exit(-1);
    }
}

int main()
{
    InitTime(2, 0);
    while(1)
        ;
    return 0;
}
```

附带作业（针对已经完成上课代码的同学）

A，B 两个进程通过管道通信，像以前的互相聊天一样，然后 A 进程每次接收到的数据通过 A1 进程显示（一个新进程，用于显示 A 接收到的信息），A 和 A1 间的数据传递采用共享内存，对应的有一个 B1 进程，用于显示 B 进程接收到的信息。针对 A，B 进程，退出时采用 ctrl+c 退出，当收到对应信号后，自身进程能够通过信号处理函数进行资源清理，清理后 exit 退出进程。（A1，B1，手动关闭即可）。界面图如下。


A1进程，显示窗口
B1进程，显示窗口
A进程，输入窗口
B进程，输入窗口