7 信号

7.1 信号的概念

在进程运行过程中，经常会产生一些事件，这些事件的产生和进程的执行往往是异步的，信号提供了一种在软件层面上的异步处理事件的机制。在硬件层面上的异步处理机制是中断，信号是中断的软件模拟，是一种**软件中断**。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。

```
$kill -l
$man 7 signal
```

7.2 信号的产生

产生的信号的方法有很多种：

- 用户按下`ctrl+c`可以产生一个SIGINT中断信号。（这种信号的产生过程：首先硬件上触发一个中断，操作系统需要切换到内核态来执行中断处理程序，在中断处理程序当中会产生一个信号，再由进程处理信号的递送）
- 进程出现除0、访问异常的内存位置时会触发信号，这种信号往往是由硬件检测得到并产生的。
- 用户使用`kill`命令，或者进程调用`kill`系统调用也可以产生信号，这种产生信号方式对用户的身份有所限制。
- 进程也会收到由于软件检测得到并产生信号，比如网络传输异常SIGURG、管道异常SIGPIPE和时钟异常SIGALRM。

7.3 信号的处理

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：进程接收信号后以默认的方式处理。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（ 对应的信号处理函数使用是 `signal(SIGINT,SIG_DFL)` ）。默认处理有5种可能：Term表示终止进程、Ign表示忽略信号、Core表示终止进程并产生core文件、Stop表示暂停进程、Cont表示继续进程
- 忽略信号：进程可以通过代码，显示地忽略某个信号的处理。比如如果将SIGSEGV信号进行忽略（使用信号处理函数`signal(SIGSEGV,SIG_IGN)`），这是程序运行如果访问到空指针，就不会报错了。但是某些信号比如SIGKILL是不能被忽略的
- 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。

7.4 Linux中所有信号

```
              Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
──┐
      SIGHUP        1       Term    连接断开
      SIGINT        2       Term    键盘中断
      SIGQUIT       3       Core    键盘退出
      SIGILL        4       Core    CPU指令译码阶段无法识别
      SIGABRT       6       Core    异常终止
      SIGFPE        8       Core    浮点异常（比如除0操作）
      SIGKILL       9       Term    终止
      SIGSEGV      11       Core    异常内存访问
      SIGPIPE      13       Term    写入无读端的管道
      SIGALRM      14       Term    定时器超时
      SIGTERM      15       Term    终止
      SIGUSR1   30,10,16    Term    自定义信号1
      SIGUSR2   31,12,17    Term    自定义信号2
      SIGCHLD   20,17,18    Ign     子进程终止或者暂停
      SIGCONT   19,18,25    Cont    暂停后恢复
      SIGSTOP   17,19,23    Stop    暂停
```

```
SIGTSTP    18,20,24    Stop    终端输入的暂停
SIGTTIN    21,21,26    Stop    后台进程控制终端读
SIGTTOU    22,22,27    Stop    后台进程控制终端写
```

7.5 信号的实现机制

尽管信号有着多种产生来源，但是对于进程而言，信号的产生只不过是修改了内核的`task_struct`结构体的一些表示信号的成员，就是说，信号产生于内核。

当一个进程处于一个可以接受信号状态的时候（这种状态被称为**响应时机**），它会取得信号，并执行默认行为、忽略或者是自定义信号处理函数。因此信号的实现可以分为两个阶段，**信号产生**表示内核已知信号发生并修改内核数据结构的内容；**信号递送**表示内核执行信号处理流程。已经产生但是还没有传递的信号被称为**挂起信号**（pending signal）或者是未决信号。如果信号一直处于未决状态，那么就称进程**阻塞**了信号传递。

一个信号的产生对会对进程产生什么样的影响会由两个重要的位图来管理。位图`sigblock`集合（也叫做**信号屏蔽字**）决定了产生的信号是否会被阻塞；位图`sigpending`集合集中管理了所有的挂起信号。

由进程的某个操作产生的信号称为**同步信号**(synchronous signals)，例如在代码中除 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)。同步和异步是编程当中一个非常重要的概念，同步表示事件之间的执行顺序是确定的，这种事件处理方式就更符合人类的认知习惯；异步表示事件之间的执行顺序是随机的，异步模式在使用良好的情况下更符合真实的物理世界，也能实现跟高的执行效率。目前，很多框架都是采用异步的方式实现底层请求处理，但是框架实现了良好的封装，这样程序员可以比较容易地用同步的方式编写程序代码，间接地使用异常方式提高运行效率。

用户使用自定义信号处理函数的主要目的就是实现**进程的有序退出**。如果没有实现进程的有序退出，就产生一些严重的运行事故，比如著名的“东京证券交易所系统宕机事件”。

7.6 函数`signal`注册信号

`signal`函数可以用来捕获信号并且指定对应的信号处理行为。

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
//这个函数是由ISO C定义的，因为要支持多个平台，所以它对信号的支持功能非常有限
```

`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，表示捕获信号后执行的函数（这里就是一个回调函数），该参数也可以是 SIG_DFL，表示交由系统缺省处理，或者也可以是 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，是一个回调函数，在`signal` 函数中注册，注册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回调函数。该函数只有一个整型参数，表示信号值。

```
#include <func.h>
void sigfunc(int signum){
    printf("signum = %d is coming\n",signum);//signum表示信号的具体数
值
}
int main()
{
    signal(SIGINT, sigfunc);//将SIGINT信号的处理行为注册成sigfunc
    printf("proces begin!\n");
    while(1);
    return 0;
}
```

启动进程以后，如果向这个进程发送键盘中断，可以看到进程不会终止，反而会调用回调函数打印一些信息。所以从这里可以了解到bash进程的实现原理，bash进程会注册SIGINT信号，这样当从键盘输入中断时，bash进程不会终止了。

7.6.1 信号递送的流程

回调函数的代码段是由用户自己编写的，它在内存中处于用户态的代码段当中，但是执行这部分代码和普通的函数调用有着显著的区别。

这里简单地描述一下信号递送的内核流程：

- 进程执行过程中调用了`signal`函数（或者随后要讲解的`sigaction`函数）为信号递送注册将要执行的回调函数。
- 进程执行过程由于各种原因从用户态切换到内核态（比如本进程调用系统调用、外界产生的硬件中断）
- 在处理完中断或者异常以后，进程将要恢复到用户态，此时需要检查一下`sigblock`和`sigpending`，检查是否存在可以被递送的信号产生。

- 如果内核发现进程`task_struct`的`sigpending`中存在任意一个可以被递送的信号（这个进程可以是不占用CPU的进程），这个时候内核决定不再返回原来进程，而是转而执行对应的回调函数。此时CPU处于**中断上下文**的状态，PC寄存器中存储了回调函数代码段的首地址，回调函数也拥有了和原进程执行流独立的用户态栈。
- 当回调函数执行完毕以后，CPU会切换回内核态，并且将上下文恢复为进程上下文，从而继续进程的正常执行。
- stdout对于进程上下文和中断上下文都是全局的，所以无论是进程正常执行还是回调函数执行都可以使用stdout。

7.7 注册多个信号

使用`signal`函数是可以同时注册多个信号的。

```
#include <func.h>
void sigfunc(int signum){
    printf("signum = %d is coming\n",signum);//signum表示信号的具体数值
}
int main()
{
    signal(SIGINT, sigfunc);//将SIGINT信号的处理行为注册成sigfunc
    signal(SIGQUIT, SIG_IGN);
    printf("proces begin!\n");
    while(1);
    return 0;
}
```

此时无论是输入键盘中断还是键盘终止都无法退出进程了。

7.8 内核不可中断状态

处于内核不可中断状态（进程控制块的state成员此时为TASK_UNINTERRUPTIBLE）的进程无法接受并处理信号。处于这种的状态的进程在`ps`中显示为D，通常这种状态出现在进程必须不受干扰地等待或者等待事件很快会发生的时候出现，比如进程正在等待磁盘读写数据的时候。对于非嵌入式程序员而言，这种状态是几乎没办法实现。内核不可中断状态是进程等待态的一种形式。

7.9 多个信号处理同时执行

在使用函数`signal`时，如果进程收到一个信号，自然地就会进入信号处理的流程，如果在信号处理的过程中：

- 接受到了另一个不同类型信号。那么当前的信号处理流程是会被中断的，CPU会先转移执行新到来的信号处理流程，执行完毕以后再恢复原来信号的处理流程。
- 接受到了另一个相同类型信号。那么当前的信号处理流程是会不会被中断的，CPU会继续原来的信号处理流程，执行完毕以后再响应新来到的信号。
- 如果接受到了连续重复的相同类型的信号，后面重复的信号会被忽略，从而该信号处理流程只能再多执行一次。

```
#include <func.h>
void sigfunc(int signum){
    printf("before signum = %d is coming\n",signum);
    sleep(2);
    printf("after signum = %d is coming\n",signum);
}
int main()
{
    signal(SIGINT, sigfunc);//将SIGINT信号的处理行为注册成sigfunc
    signal(SIGQUIT, sigfunc);
    printf("proces begin!\n");
    while(1);
    return 0;
}
```

当进程处于某个信号处理流程的时候，如果再产生一个同类型信号，信号处理流程不会中断（因为回调函数执行过程中，信号sigblock会将本信号对应位设置为1），而sigpending中的对应位会设置为1，表示此时存在一个挂起信号，随后产生的同类型信号将不再被记录。之所以设计同类信号无法中断，是考虑到信号处理流程可能会修改静态数据或者堆数据（比如stdout），这种函数被称为**不可重入**函数，如果中断处理流程，可能会导致进程破坏这些数据。

7.10 重新注册信号处理流程

如果希望在进程执行过程中重新指定信号的处理流程，可以多次使用`signal`函数。

```
#include <func.h>
int main()
{
    signal(SIGINT, SIG_IGN);
    printf("proces begin!\n");
    sleep(10);
    printf("sleep over!\n");
    signal(SIGINT, SIG_DFL);
    while(1);
    return 0;
}//当进程sleep结束以后，进程就可以接受到键盘中断了。
```

7.11 函数`sigaction`注册信号

在 `signal` 处理机制下，在一些特殊的场景，它满足这样的行为：

- 注册一个信号处理函数，并且处理完毕一个信号之后，不需要重新注册，就能够捕捉下一个信号。
- 如果信号处理函数正在处理信号，并且还没有处理完毕时，又产生了一个同类型的信号，那么会依次处理信号，并且忽略多余的信号。
- 如果信号处理函数正在处理信号，并且还没有处理完毕时，又产生了一个不同类型的信号，那么会中断当前处理流程，跳转新信号的处理流程。
- 如果程序阻塞在一个系统调用（比如 `read`）时，产生一个信号，这时会有两种不同类型的行为，一种大多数系统调用的行为，例如读写磁盘文件时的等待，信号递送时会让系统调用返回错误再接着进入信号处理函数；另一种是先跳转到信号处理函数，等信号处理完毕后，再重新启动系统调用，这些系统调用往往是低速系统调用，比如读写管道、终端和网络设备，又比如`wait`和`waitpid`等等。

显然如果使用`signal`函数，在这些场景下的执行流程是固定的并且无法调整的，而使用函数`sigaction`就可以自定义这些场景下进程的行为。

```
include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
struct sigaction {
               void     (*sa_handler)(int);
               void     (*sa_sigaction)(int, siginfo_t *, void *);
               sigset_t   sa_mask;
               int        sa_flags;
               void     (*sa_restorer)(void);
};
```

oldact参数表示之前的回调函数，通常会传入空指针，所以我们主要关心act参数。

- sa_handler成员或者sa_sigaction成员（一般使用sa_sigaction）是用来描述信号处理的回调函数。
- 如果回调函数不是SIG_IGN或者SIG_DFL时，sa_mask成员描述了一个信号集，调用回调函数以前，该信号集要加入进程的信号屏蔽字中，回调函数返回的时候再恢复原来的信号屏蔽字，除此以外，正在处理的信号默认是被阻塞的。
- sa_flags参数表示信号处理方式掩码，可以用来设置信号的处理模式。
- sa_restorer成员暂时无用。

7.11.1 sa_flags的取值集合

取值	含义
SA_SIGINFO	表示选择sa_sigaction而不是sa_handler作为回调函数
SA_RESETHAND	处理完捕获的信号以后，信号处理回归到默认，使用情况较少
SA_NODEFER	解除所有阻塞行为。特别地，执行信号处理流程可以处理同类信号传递，按照栈的方式执行。
SA_RESTART	让低速系统调用可以自动重启

```
//SA_SIGINFO
//SA_RESETHAND
#include <func.h>
void sigfunc(int signum, siginfo_t *p, void *p1){
  //printf("%d is coming", signum);
  printf("%d is coming\n", signum);//必须添加\n
```

```
}
int main()
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    //act.sa_flags = SA_SIGINFO;
    act.sa_flags = SA_SIGINFO|SA_RESETHAND;
    act.sa_sigaction = sigfunc;
    int ret = sigaction(SIGINT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    while(1);
    return 0;
}
```

```
//SA_NODEFER
#include <func.h>
void sigfunc(int signum, siginfo_t *p, void *p1){
    printf("before %d is coming\n", signum);//必须添加\n
    sleep(3);
    printf("after %d is coming\n", signum);//必须添加\n
}
int main()
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    //act.sa_flags = SA_SIGINFO;
    act.sa_flags = SA_SIGINFO|SA_NODEFER;
    act.sa_sigaction = sigfunc;
    int ret = sigaction(SIGINT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    while(1);
    return 0;
}//程序的运行流程和sleep实现有关。
```

低速系统调用的中断处理流程，首先是使用`signal`的处理流程，在这种情况系统调用会自动重启。

```c
//使用ps命令查看进程状态时，进程阻塞在wait_w
#include <func.h>
void sigfunc(int signum){
    printf("signum = %d is coming\n",signum);
}
int main()
{
    signal(SIGINT,sigfunc);
    char buf[128] = {0};
    read(STDIN_FILENO,buf,sizeof(buf));
    puts(buf);
    return 0;
}
```

如果使用`sigaction`的处理流程，那么系统调用会出错并终止。

```c
#include <func.h>
void sigfunc(int signum, siginfo_t *p, void *p1){
    printf("%d is coming\n", signum);//必须添加\n
}
int main()
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = sigfunc;
    int ret = sigaction(SIGINT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    char buf[128] = {0};
    ret = read(STDIN_FILENO,buf,sizeof(buf));
    ERROR_CHECK(ret,-1,"read");
    return 0;
}//read: Interrupted system call
```

增加SA_RESTART可以自动重启低速系统调用。

```c
#include <func.h>
void sigfunc(int signum, siginfo_t *p, void *p1){
```

```
    printf("%d is coming\n", signum);//必须添加\n
}
int main()
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    act.sa_flags = SA_SIGINFO|SA_RESTART;
    act.sa_sigaction = sigfunc;
    int ret = sigaction(SIGINT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    char buf[128] = {0};
    ret = read(STDIN_FILENO,buf,sizeof(buf));
    ERROR_CHECK(ret,-1,"read");
    return 0;
}
```

7.11.2 sa_mask设置阻塞集合

在信号处理流程中，如果递送了新的不同类型信号，在没有指定SA_NODEFER参数的情况下，新信号将会中断正在执行的信号处理流程。为了避免这种中断行为，可以使用sa_mask参数来增加一些信号的阻塞操作。

```
typedef struct
{
  unsigned long int __val[(1024 / (8 * sizeof (unsigned long
int)))];
} __sigset_t;
typedef __sigset_t sigset_t;
//sigset_t的本质就是一个位图，共有1024位
```

```c
#include <signal.h>
int sigemptyset(sigset_t *set);
//初始化信号集，清除所有信号
int sigfillset(sigset_t *set);
//初始化信号集，包括所有信号
int sigaddset(sigset_t *set, int signum);
//增加信号
int sigdelset(sigset_t *set, int signum);
//删除信号
int sigismember(const sigset_t *set, int signum);
//检查信号处于信号集之中
```

```c
#include <func.h>
void sigfunc(int signum, siginfo_t *p, void *p1){
    printf("before %d is coming\n", signum);
    sleep(3);
    printf("after %d is coming\n", signum);
}
int main()
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = sigfunc;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask,SIGQUIT);
    int ret = sigaction(SIGINT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    ret = sigaction(SIGQUIT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    while(1);
    return 0;
}
```

需要特别注意的是阻塞屏蔽和忽略信号有着既然不同的含义，阻塞表示信号产生了但是还未递送，内核会维护一个所有未决信号的位图，如果信号已经被阻塞，再次产生信号就会被忽略了。被阻塞的信号将会后续执行，而被忽略的信号就被丢弃了。

7.12 系统调用sigpending

使用系统调用sigpending可以获取当前所有未决信号（已经产生没有递送的信号）的集合。通常这个系统调用是在回调函数当中使用的，用于检查当前是否阻塞了某个信号。

```
#include <func.h>
void sigfunc(int signum, siginfo_t *p, void *p1){
    printf("before %d is coming\n", signum);
    sleep(3);
    sigset_t pendingSet;
    sigpending(&pendingSet);
    if(sigismember(&pendingSet,SIGQUIT)){
        printf("SIGQUIT is pending!\n");
    }
    else{
        printf("SIGQUIT is not pending!\n");
    }
    printf("after %d is coming\n", signum);
}
int main()
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = sigfunc;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask,SIGQUIT);
    int ret = sigaction(SIGINT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    ret = sigaction(SIGQUIT,&act,NULL);
    ERROR_CHECK(ret,-1,"sigaction");
    while(1);
    return 0;
}
```

7.13 `sigprocmask`实现全程阻塞

假设存在这样一种场景，我们需要在进程中写入共享资源，自然就会采用加锁/解锁操作，如果这种写入过程十分重要，那么我们往往需要在加解锁之间屏蔽某些信号的递送。我们之前的阻塞操作只能在某个信号处理过程中去阻塞另一个信号，另一种解决方案的实现思路是，在加锁的时候，将信号注册为忽略，在解锁的时候将信号注册为默认。但是这样的实现方法并不是阻塞信号而是忽略信号，在访问共享资源过程中产生的信号就被丢弃掉了。

使用系统调用`sigprocmask`可以实现全程阻塞的效果。它可以检测或者更改信号屏蔽字的内容。参数how描述了如何修改；参数set指向了信号集；如果oldset非空时，则会返回当前信号屏蔽字。

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

HOW	效果
SIG_BLOCK	新的屏蔽字是set和当前屏蔽字的并集
SIG_UNBLOCK	新的屏蔽字是set的补集和当前屏蔽字的交集
SIG_SETMASK	新的屏蔽字是set

```
#include <func.h>

int main()
{
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask,SIGINT);
    int ret = sigprocmask(SIG_BLOCK,&mask,NULL);
    ERROR_CHECK(ret,-1,"sigprocmask");
    printf("block success!\n");
    sleep(3);
    ret = sigprocmask(SIG_UNBLOCK,&mask,NULL);
    ERROR_CHECK(ret,-1,"sigprocmask");
        while(1);
    return 0;
}
```

`sigpending`可以和`sigprocmask`之间配合使用：

```
#include <func.h>

int main()
{
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask,SIGINT);
    int ret = sigprocmask(SIG_BLOCK,&mask,NULL);
    ERROR_CHECK(ret,-1,"sigprocmask");
    printf("block success!\n");
    sleep(3);
    sigset_t pend;
    sigemptyset(&pend);
    sigpending(&pend);
    if(sigismember(&pend,SIGINT)){
        printf("SIGINT is pending!\n");
    }
    else {
        printf("SIGINT is not pending!\n");
    }
    ret = sigprocmask(SIG_UNBLOCK,&mask,NULL);
    ERROR_CHECK(ret,-1,"sigprocmask");
    while(1);
    return 0;
}
```

7.14 `kill`系统调用

系统调用`kill`可以用来给另一个进程发送信号。pid参数表示进程ID，sig参数表示信号数值，pid如果是-1，表示给所有可以发送信号的进程发送信号，如果小于-1，则根据其绝对值，去关闭其作为组长的进程组。

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

```c
#include <func.h>
int main(int argc, char *argv[])
{
    // ./kill -9 pid
    ARGS_CHECK(argc,3);
    int sig = atoi(argv[1]+1);
    pid_t pid = atoi(argv[2]);
    int ret = kill(pid,sig);
    ERROR_CHECK(ret,-1,"kill");
    return 0;
}
```

7.15 pause和sigsuspend

pause系统调用可以用来阻塞一个进程，直到某个信号被递送时，进程会解除阻塞，然后终止进程或者执行信号处理函数。

```c
#include <unistd.h>
int pause(void);
```

如果使用sigprocmask，可以实现所谓的信号保护临界区，在临界区当中执行代码的时候，此时产生的信号将会被阻塞，临界区结束的位置只需要再使用sigprocmask即可。如果希望在临界区之后再次捕获信号，使用系统调用pause可以进行捕获，但是在临界区间产生的信号却会直接执行信号处理流程，无法使pause函数就绪。

无法捕获信号是因为信号在解除阻塞后会立即进入信号处理流程，此时进程还未进入到pause的阻塞状态中。所以为了捕获期间产生的信号，一种策略就是将解除阻塞和等待信号合并成一个原子操作，这就是sigsuspend。

```c
#include <func.h>
#include <signal.h>
void sigfunc(int signum, siginfo_t* p, void *p1){
    printf("%d is processing!\n",signum);
}
int main()
{
    struct sigaction act;
```

```
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask,SIGQUIT);
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = sigfunc;
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask,SIGINT);
    sigprocmask(SIG_BLOCK,&mask,NULL);
    for(int i = 0; i < 3; ++i){
        sleep(1);
        printf("i = %d\n",i);
    }
    sigaction(SIGINT,&act,NULL);
#if 1
    sigprocmask(SIG_UNBLOCK,&mask,NULL);
    pause();//无法就绪
#else
    sigset_t waitset;
    sigemptyset(&waitset);
    sigsuspend(&waitset);//使用sigsuspend会捕获临界区当中的信号
#endif
    return 0;
}
```

7.16 `sleep`和系统时间

7.16.1 `sleep`的一种实现方法

系统调用`alarm`可以用来实现延时信号，它的原理是设置一个定时器，将来某个时刻定时器会超时，此时会递送一个SIGALRM信号。这里要注意的是每个进程只有一个闹钟时间，所以重复使用`alarm`会更新超时时间，并且返回值会是原来闹钟时间的剩余秒数。

```c
#include <func.h>
int main()
{
    int ret = alarm(3);
    printf("first ret = %d\n", ret);
    sleep(2);
    ret = alarm(2);
    printf("second ret = %d\n", ret);
    return 0;

}
```

利用`alarm`系统调用和`pause`系统调用，我们可以比较简陋地实现`sleep`功能，不过如果`alarm`和`pause`存在进程切换，导致执行`pause`之前，信号就已经递送，那么将会导致永远挂起。使用`sigsuspend`可以解决这个问题。

```c
#include <func.h>

void sigfunc(int signum){
    //nothing to do

}
void mySleep(int i){
    struct sigaction act;
    act.sa_handler = sigfunc;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGALRM,&act,NULL);

    sigset_t blkmask;
    sigemptyset(&blkmask);
    sigaddset(&blkmask,SIGALRM);
    sigprocmask(SIG_BLOCK,&blkmask,NULL);

    alarm(i);
    sigdelset(&blkmask,SIGALRM);
    sigsuspend(&blkmask);
```

```
}
int main()
{
    time_t before = time(NULL);
    mySleep(1);
    mySleep(2);
    mySleep(3);
    time_t now = time(NULL);
    printf("time = %ld\n",now - before);
    return 0;

}
```

当然，POSIX标准并没有规定`sleep`采用哪种具体实现，尽管目前Linux普遍采用`nanosleep`实现`sleep`，但是为了兼容性，使用`sleep`的时候不用混用`alarm`。

7.17 时钟处理

`setitimer`系统调用负责调整间隔定时器。间隔定时器在创建的时候，就会设置一个时间间隔，定时器到达时间间隔时，调用进程会产生一个信号，随后定时器被重置。

操作系统为每个进程维护3种不同的定时器，分别是真实计时器、虚拟计时器和实用计时器。

- 真实计时器会记录真实的时间（也就是时钟时间），当时间到时，会产生一个SIGALRM信号。
- 虚拟计时器会记录用户态模式下的CPU时间，当时间到的时候，会产生一个SIGVTALRM信号。
- 实用计时器会记录用户态以及内核态模式下的CPU时间，当时间到的时候，会产生一个SIGPROF信号。

使用`fork`的时候子进程不会继承父进程的定时器，使用`exec`时候，定时器不会销毁。不要混用`setitimer`、`sleep`、`usleep`和`alarm`。

```c
 #include <sys/time.h>
int getitimer(int which, struct itimerval *curr_value);·
int setitimer(int which, const struct itimerval *new_value,
                      struct itimerval *old_value);
struct itimerval {
               struct timeval it_interval; /* Interval for periodic
timer */
               struct timeval it_value;    /* Time until next
expiration */
           };

struct timeval {
               time_t      tv_sec;         /* seconds */
               suseconds_t tv_usec;        /* microseconds */
           };
```

```c
#include <func.h>
void sigfunc(int signum){
    time_t now;
    time(&now);
    printf("%s\n",ctime(&now));

}
int main(){
    struct sigaction act;
    act.sa_handler = sigfunc;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGALRM,&act,NULL);
    struct itimerval timer;
    timer.it_value.tv_sec = 1;
    timer.it_value.tv_usec = 0;
    timer.it_interval.tv_sec = 3;
    timer.it_interval.tv_usec = 0;
    setitimer(ITIMER_REAL, &timer, NULL);
    sigfunc(0);
    //sleep(2);sleep对结果没有影响
    while(1);
```

```
}
```

```
#include <func.h>
void sigfunc(int signum){
    time_t now;
    time(&now);
    printf("%s\n",ctime(&now));

}
int main(){
    struct sigaction act;
    act.sa_handler = sigfunc;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGPROF,&act,NULL);
    struct itimerval timer;
    timer.it_value.tv_sec = 1;
    timer.it_value.tv_usec = 0;
    timer.it_interval.tv_sec = 3;
    timer.it_interval.tv_usec = 0;
    setitimer(ITIMER_PROF, &timer, NULL);
    sigfunc(0);
    sleep(2);//这里会有不同
    while(1);
}
```

如果使用实用计时器的同时，如果此时有其他进程大量占用CPU，那么统计的间隔时间会大大提升。

小项目 四窗口聊天

在四个终端上分别启动A、A1、B、B1四个进程。

- A进程和B进程用来输入聊天数据
- A1进程负责显示B发送给A的消息，B1进程负责显示A发送给B的消息。

- A、B两个进程使用有名管道通信，和即时聊天一致。
- A 和 A1 间的数据传递采用共享内存
- B和B1之间采用消息队列传递消息
- 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
- 单个进程退出以后，需要发送信号给其他进程，让它们也能有序退出

对于A1进程而言，它需要能够显示出来自B由A转发的消息，还需要能够处理A发送过来的信号，故它需要能够区分出这两种行为。这里可以采取一定的**协议**设计。比如把数据的前面若干字节专门用来存储数据类型的信息，比如0表示发送的是PID（当然实际应用可以用宏或者枚举），1表示发送的是消息，2表示发送的是控制信息等等。