第3章 传统的进程间通信

实验目的

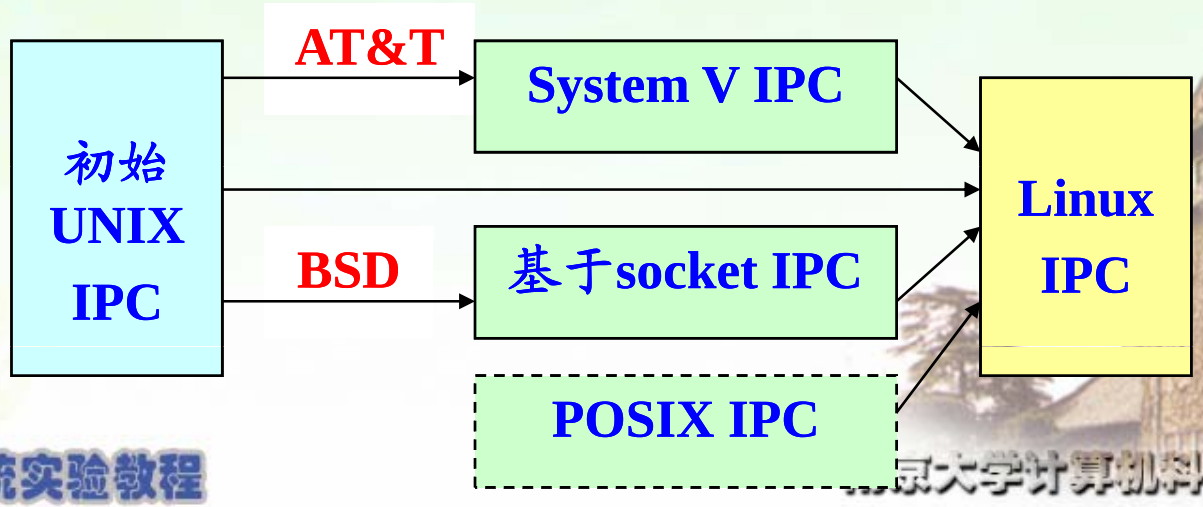
- 理解信号和管道的概念及用于实现进程通信的原理
- 掌握信号通信机制，实现进程之间通过信号进行通信
- 掌握匿名管道及有名管道通信机制，实现进程之间通过管道进行通信

主要内容

- 背景知识
  - 进程间的通信方式
  - 信号通信
  - 管道通信
- 实验内容
  - 信号通信
  - 匿名管道通信
  - 命名管道通信
  - 使用命名管道建立客户/服务器关联程序

Linux进程通信的主要手段

- 基本上从Unix平台上的进程通信手段继承而来
  - AT&T的贝尔实验室
    - ✓system V IPC：单机内进程间通信
  - BSD（加州大学伯克利分校的伯克利软件发布中心）
    - ✓套接字（socket）进程间通信机制
  - Linux
    - ✓继承两者



进程通信作用

- 共享资源
  - 保护临界资源，支持互斥访问
  - 进程同步
- 协同工作
  - 数据共享
  - 消息通知
  - 数据交换

Linux进程通信主要手段

- 传统信号通信
  - 信号（signal）：进程或内核均使用信号通知一个进程有某种事件发生
  - 管道（pipe）及有名管道（named pipe）：用于具有亲缘关系的进程之间的通信
- System V IPC进程通信
  - 消息（message）队列：是消息的链接表，在进程之间以传递消息形式进行通信
  - 共享主存(shared memory)：支持多个进程可以访问同一块主存空间
  - 信号量（semaphore）：用做用户空间的进程之间及同一进程内不同线程之间的同步手段
- 套接字（socket）

主要内容

- 背景知识
  - 进程间的通信方式
  - 信号通信
  - 管道通信
- 实验内容
  - 信号通信
  - 匿名管道通信
  - 命名管道通信
  - 使用命名管道建立客户/服务器关联程序

信号

- 对中断的模拟，也称软中断信号或软中断
- 主要用于向进程发送异步事件信号，如
  - 键盘中断可能产生信号
  - 浮点运算溢出或者内存访问错误等也可产生信号
  - shell 通常利用信号向子进程发送作业控制命令
- 信号事件来源
  - 硬件来源
    - ✓ 硬件操作，如按Ctrl+C
    - ✓ 硬件故障
  - 软件来源
    - ✓ 常用信号发送函数，如kill（）,raise（）,alarm
    - ✓ 非法运算

信号分类

- 不可靠信号
  - 早期机制上的信号称为“不可靠信号”，信号值小于SIGRTMIN
  - 进程每次处理信号后，就将对信号的响应设置为默认动作，需要调用signal( ) 重新安装该信号
  - 信号可能丢失
  - 非实时信号属于不可靠信号，支持排队
- 可靠信号
  - 信号值位于SIGRTMIN和SIGRTMAX之间
  - 支持排队，不会丢失
  - 新信号安装函数sigaction( )和信号发送函数sigqueue( )
  - 实时信号都是可靠信号，不支持排队
- signal( )与sigaction( )的区别
  - sigaction( )能传递信息给信号处理函数
  - signal( )安装的信号不能向信号处理函数传递信息
  - 对于信号发送函数来说也是一样

常见信号

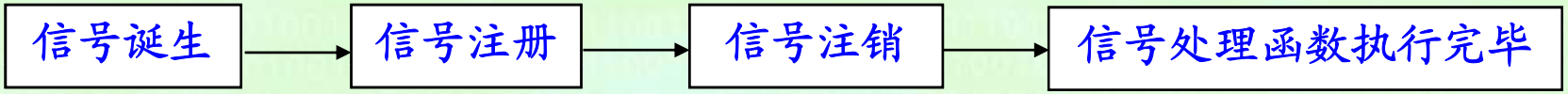
❖ **Kill –l**可列出所有信号

值	C 语言宏	用途
1	SIGHUP	从终端上发出的结束信号
2	SIGINT	来自键盘的中断信号（Ctrl-c）
3	SIGQUIT	来自键盘的退出信号（Ctrl-\）
8	SIGFPE	浮点异常信号（例如浮点运算溢出）
9	SIGKILL	该信号结束接收信号的进程
14	SIGALRM	进程的定时器到期时，发送该信号
15	SIGTERM	kill 命令发出的信号
17	SIGCHLD	标识子进程停止或结束的信号
19	SIGSTOP	来自键盘(Ctrl-z)或调试程序的停止执行信号

进程对信号的响应方式

- 忽略信号
  - 进程忽略接收到的信号，不做任何处理，象未发生过一样
  - SIGKILL和SIGSTOP信号不能忽略
- 捕获信号
  - 类似中断处理程序，进程本身可在系统中为需要处理的信号定义信号处理函数
  - 一旦相应信号发生，执行对应的信号处理函数
- 缺省操作
  - 信号由内核的默认处理程序处理
  - Linux为每种信号都定义默认操作
  - 进程可通过signal( )来指定进程对某个信号的处理行为

信号生命周期



- 信号诞生
  - 指触发信号的事件发生
- 信号注册
  - 将信号值加入到进程的未决信号集，并将信号所携带的信息保存到未决信号信息链的某个sigqueue结构中
    - ✓ 对于实时信号，不管该信号是否已经在进程中注册，都会被再注册一次
    - ✓ 对非实时信号，如果该信号已经在进程中注册，则该信号将被丢弃
- 信号注销
  - 目标进程每次从核心空间返回到用户空间时都会检测是否有信号等待处理
  - 如果存在未决信号等待处理且该信号没有被进程阻塞，把信号在未决信号链中占有的结构卸掉（对实时进程只删除一个sigqueue结构）
- 信号生命终止
  - 执行相应的信号处理函数

task_struct中与信号处理相关的成员

- **struct sigpending pending**
  - 维护本进程中的未决信号，包括当前挂起的信号及当前阻塞的信号

```
struct sigpending {
    struct sigqueue *head, **tail;
    sigset_t signal;
};
```

```
struct sigqueue {
    struct sigqueue *next;
    siginfo_t info;
}
```

信号处理函数

- 信号安装函数
  - signal(), sigaction()
- 信号发送函数
  - kill(), raise(), alarm(), setitimer(), pause()
- 信号操作函数
  - sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember()

信号安装函数—signal()

- 原型定义
  - void (*signal(int signum, void (*handler)(int)))(int);
- 参数说明
  - signum: 需设置处理方法的信号
  - handler: 处理函数，也可以是SIG_IGN或SIG_DFL。
    - ✓SIG_IGN：使用忽略该信号的操作函数
    - ✓SIG_DFL：使用缺省的信号操作函数
- 返回值
  - 调用成功，返回最后一次为安装信号signum而调用signal()时的handler值
  - 失败则返回SIG_ERR

信号安装函数—signal()示例

- **signaltest.c**
  - 信号处理函数定义

```
/*信号处理函数，其中dunno将会得到信号的值*/
void sigroutine(int dunno) {
  switch (dunno) {
  case 1:
    printf("Get a signal -- SIGHUP ");
    break;
  case 2:
    printf("Get a signal -- SIGINT ");
    break;
  case 3:
    printf("Get a signal -- SIGQUIT ");
    break;
  }
  return;
}
```

信号安装函数—signal()示例

- **signaltest.c**
  - 主程序
    - ✓SIGINT由按下键盘Ctrl+C发出
    - ✓SIGQUIT由按下键盘Ctrl+\组合键发出

```
int main( ) {
   printf("process id is %d ", getpid( ));
   /*下面设置3个信号处理方法 */
   signal(SIGHUP, sigroutine);
   signal(SIGINT, sigroutine);
   signal(SIGQUIT, sigroutine);
   for (;;);
}
```

信号安装函数—signal()示例

- **signaltest.c**
  - 运行结果

```
localhost:~$ ./sigaltest
process id is 463
Get a signal -SIGINT /*按下Ctrl+C得到的结果 */
Get a signal –SIGQUIT/*按下Ctrl+\组合键得到的结果 */
[1]+ Stopped ./signaltest
localhost:~$ bg
[1]+ ./signaltest &
localhost:~$ kill -HUP 463 /*向进程发送SIGHUP信号 */
localhost:~$ Get a signal – SIGHUP
kill -9 463  /*向进程发送SIGKILL信号，终止进程 */
localhost:~$
```

信号安装函数—sigaction()

- 原型定义
  - int sigaction(int signum, const struct sigaction *act, struct sigaction *oact);
- 参数说明
  - signum: 要设置处理方法的信号
  - act
    - ✓对该信号进行如何处理的信息
    - ✓为空时，进程以缺省方式对信号处理
  - oact: 保存原来对这个函数的处理信息，一般选NULL
- 说明
  - SIGKILL和SIGSTOP不能通过sigaction注册使用
  - act和oact为空时，将检查信号的有效性

struct sigaction结构

```
struct sigaction {
    union {
        __sighandler_t _sa_handler;
        void (*_sa_sigaction)(int, struct siginfo *, void *);
    } _u;
    sigset_t sa_mask;
    unsigned long sa_flags;
    void (*sa_restorer)(void);
};
```

❖ 说明

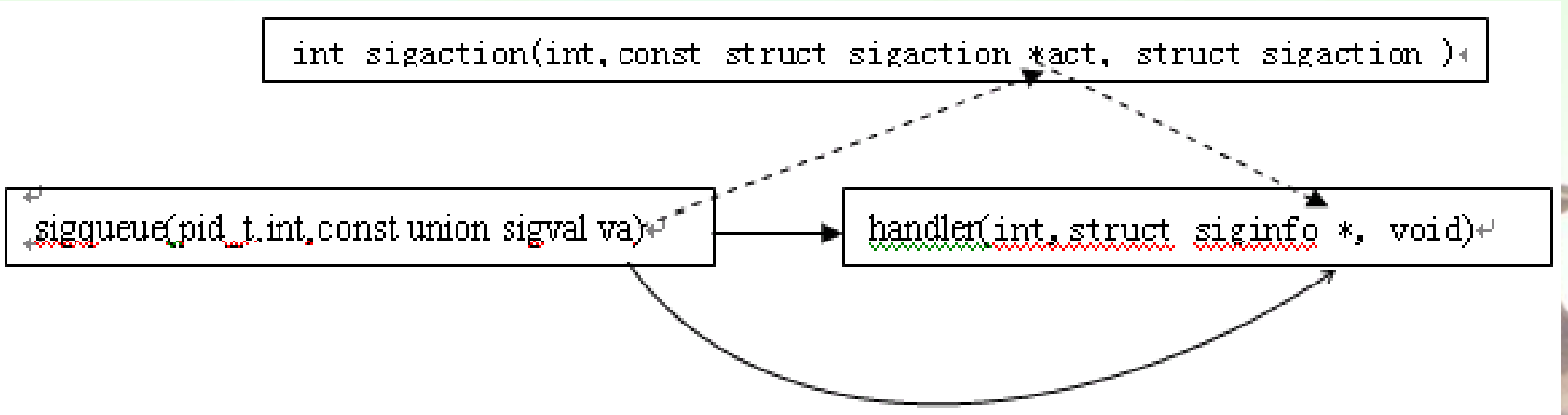
- _sa_handler以及*_sa_sigaction指定信号处理函数,也可为SIG_DFL(采用缺省的处理方式)或为SIG_IGN（忽略信号）。
  - ✓ _sa_handler指定的处理函数只有一个参数，即信号值，所以信号不能传递除信号值之外的任何信息
  - ✓ sa_sigaction的原型是一个带三个参数，第一个参数为信号值，第二个参数是一个指向struct siginfo结构的指针，第三个参数没有使用

struct sigaction结构

- 说明
  - sa_mask指定在信号处理程序执行过程中，哪些信号应当被阻塞。默认当前信号本身被阻塞
  - sa_flags包含了许多标志位
    - ✓SA_SIGINFO：表示信号附带的参数可以传递到信号处理函数中
    - ✓即使sa_sigaction指定信号处理函数，如果不设置SA_SIGINFO，信号处理函数同样不能得到信号传递过来的数据，在信号处理函数中对这些信息的访问都将导致段错误

信号传递附加消息的流程

- 系统调用sigqueue发送信号时，sigqueue的第三个参数就是sigval联合数据结构，
- 当调用sigqueue时，该数据结构中的数据就将拷贝到信号处理函数的第二个参数中
  - 在发送信号同时，就可以让信号传递一些附加信息
  - 信号可以传递信息对程序开发是非常有意义的



信号安装函数—sigaction()示例

❖ **signalactiontest.c**

```
#define PROMPT"你想终止程序吗?"
char *prompt=PROMPT;
void ctrl_c_op(int signo){
  write(STDERR_FILENO,prompt,strlen(prompt));
}
int main( ){
  struct sigaction act;
  act.sa_handler=ctrl_c_op;
  sigemptyset(&act.sa_mask);
  act.sa_flags=0;
  if(sigaction(SIGINT,&act,NULL)<0){
    fprintf(stderr,"Install Signal Action
     Error:%s\n\a",strerror(errno));
    exit(1);
  }
  while(1);
}
```

需接收附加信息的信号处理

- 必须给sa_sigaction赋信号处理函数指针
- 同时还要给sa_flags赋SA_SIGINFO

```
void sig_handler_with_arg(int sig,siginfo_t *sig_info,void *unused)
{……}

int main(int argc,char **argv) {
  struct sigaction sig_act;
  ……
  sigemptyset(&sig_act.sa_mask);
  sig_act.sa_sigaction=sig_handler_with_arg;
  sig_act.sa_flags=SA_SIGINFO;
   ……
 }
```

信号发送函数—kill()

- 功能
  - 向进程或进程组发送一个信号
- 函数原型
  - int kill(pid_t pid, int sig);
- 参数说明
  - pid: 接收信号的进程（组）的进程号
    - pid>0：发送到进程号为pid的进程
    - pid=0：发送给当前进程所属进程组里的所有进程
    - pid=-1：发送给除了1号进程和自身以外的所有进程
    - pid<-1：发送给属于进程组-pid的所有进程
  - sig: 发送的信号
    - sig=0：不发送信号
- 返回值
  - 执行成功时，返回值为0
  - 错误时，返回-1
- 说明
  - 一个进程被允许将信号发送到进程pid时，必须拥有root权限或者是发出调用的进程的UID 或EUID与指定接收的进程的UID或保存用户ID（savedset-user-ID）相同。

信号发送函数—sigqueue()

- 功能
  - 针对实时信号提出，支持信号带有参数，与函数sigaction()配合使用。
- 函数原型
  - int sigqueue(pid_t pid, int signo, const union sigval val)
- 参数说明
  - pid: 接收信号的进程ID
  - signo: 即将发送的信号
  - val: 指定信号传递的参数，即通常所说的4字节值
- 说明
  - sigqueue()比kill()传递更多的附加信息
    - ✓ 但sigqueue()只能向一个进程发送信号。
    - ✓ 如果signo=0，将执行错误检查，不发送任何信号，常用于检查pid的有效性以及当前进程是否有权限向目标进程发送信号
  - 调用sigqueue时，sigval_t指定的信息会拷贝到3参数信号处理函数
  - sigqueue()发送非实时信号时
    - ✓ 第三个参数包含的信息仍然能够传递给信号处理函数
    - ✓ 仍然不支持排队，所有相同信号都被合并为一个信号

信号发送函数—raise()

- 功能
  - 向自身发送信号
- 函数原型
  - **int raise(int sig);**
- 参数说明
  - **sig**：发送的信号
    - ✓ **sig=0**：不发送信号
- 返回值
  - 执行成功时，返回值为**0**
  - 错误时，返回**-1**
- 说明
  - **raise( )**可以通过**kill( )**实现，**raise(signo)**等价于
    - ✓ **kill(getpid( ), signo);**

信号发送函数—alarm()

- 功能
  - 设置一个定时器，当定时器计时到达时，将发出信号SIGALRM给进程
- 函数原型
  - unsigned int alarm(unsigned int seconds);
- 参数说明
  - seconds： 等待秒数
    - ✓如果seconds为0，则之前设置的闹钟会被取消，并将剩下的时间返回。若之前未设闹钟则返回0
- 说明
  - alarm( )只设定为发送一次信号
  - 若需要重复设定定时器，则要多次使用alarm( )函数

信号发送函数—alarm()示例

❖ **alarmtest.c**

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

void my_alarm_handle(int sign_no){
if(sign_no==SIGALRM)
  printf("i have been waken up\n");
}

int main(){
printf("sleep for 5s ......\n");
signal(SIGALRM,my_alarm_handle);
alarm(5);
pause();
return 0;
}
```

信号发送函数—setitimer()

- 功能
  - 设置定时器，支持多种定时器
- 函数原型
  - int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
- struct itimerval结构定义

```
struct itimerval {
    struct timeval it_interval;    /*下一次取值*/
    struct timeval it_value;       /*本次设定值*/
};
```

- struct timeval结构定义

```
struct timeval {
    long tv_sec;   /*秒*/
    long tv_usec; /*微秒，1秒=1000000微秒*/
};
```

信号发送函数—setitimer()

- 参数说明
  - which: 逻辑定时器类型
    - ✓ ITIMER_REAL：按实际时间计时，计时到达将给进程发送SIGALRM信号
    - ✓ ITIMER_VIRTUAL：仅当进程执行时才进行计时。计时到达将发送SIGVTALRM信号给进程
    - ✓ ITIMER_PROF：当进程执行时和系统为该进程执行时都计时。与ITIMER_VIRTUAL是一对，该定时器经常用来统计进程在用户态和核心态花费的时间，计时到达将发送SIGPROF信号给进程
  - value: 用来指明定时器的时间
  - ovalue：如果不为空，则保存上次调用设定的值
- 工作机制
  - 定时器将it_value递减到0时，产生一个信号，并将it_value的值设定为it_interval的值，然后重新开始计时，如此往复
  - it_value=0时，计时器停止，或者当它计时到期，而it_interval为0时停止。
- 返回值
  - 执行成功时，返回值为0
  - 错误时，返回-1

信号发送函数—setitimer()示例

- **settimertest.c**
  - 信号处理函数

```
void sigroutine(int signo) {
   switch (signo) {
      case SIGALRM:
      printf("Catch a signal -- SIGALRM ");
      break;
   case SIGVTALRM:
      printf("Catch a signal -- SIGVTALRM ");
      break;
   }
   return;
}
```

信号发送函数—setitimer()示例

❖settimertest.c

➢ 主程序

```
int main( ){
   struct itimerval value,ovalue,value2;
   sec = 5;
   printf("process id is %d ",getpid( ));
   signal(SIGALRM, sigroutine);
   signal(SIGVTALRM, sigroutine);
   value.it_value.tv_sec = 1;
   value.it_value.tv_usec = 0;
   value.it_interval.tv_sec = 1;
   value.it_interval.tv_usec = 0;
   setitimer(ITIMER_REAL, &value, &ovalue);
   value2.it_value.tv_sec = 0;
   value2.it_value.tv_usec = 500000;
   value2.it_interval.tv_sec = 0;
   value2.it_interval.tv_usec = 500000;
   setitimer(ITIMER_VIRTUAL, &value2, &ovalue);
   for (;;) ;
}
```

信号发送函数—setitimer()示例

- **settimertest.c**
  - 运行结果
  - **localhost:~$ ./settimertest**
    process id is 579
    Catch a signal – SIGVTALRM
    Catch a signal – SIGALRM
    Catch a signal – SIGVTALRM
    Catch a signal – SIGVTALRM
    Catch a signal – SIGALRM
    Catch a signal –SIGVTALRM

信号发送函数—pause()

- 功能
  - 使当前进程暂停，进入睡眠状态，直到被信号所中断
- 函数原型
  - int pause(void);
- 返回值
  - -1
- 举例
  - 使用alarm( )和pause( )实现sleep( )功能

信号发送函数—pause()示例

```
void my_alarm_handle(int sign_no){
   if (sign_no == SIGALRM) {
      printf("I have been waken up.\n");
   }
}
int main( ){
   printf("sleep for 5s ... ...\n");
   signal(SIGALRM, my_alarm_handle);
   alarm(5);
   pause( );
   return 0 ;
}
```

信号集操作函数

- 信号集定义

```
typedef struct {
    unsigned long sig[_NSIG_WORDS];
} sigset_t
```

- 函数原型
  - int sigemptyset(sigset_t *set);
    - 初始化信号集合set，将set设置为空
  - int sigfillset(sigset_t *set);
    - 初始化信号集合，将信号集set设置为所有信号的集合
  - int sigaddset(sigset_t *set, int signo);
    - 将信号signo加入到信号集set中
  - int sigdelset(sigset_t *set,int signo);
    - 将信号signo从信号集set中删除
  - int sigismember(sigset_t *set,int signo);
    - 查询信号signo是否在信号集set中

信号操作函数—sigprocmask( )

- 功能
  - 将指定的信号集合set加入到进程的信号阻塞集合中去。
- 函数原型
  - int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
- 参数
  - how: 决定函数的操作方式
    - ✓ SIG_BLOCK：增加一个信号集合到当前进程的阻塞集合之中
    - ✓ SIG_UNBLOCK：从当前的阻塞集合之中删除一个信号集合
    - ✓ SIG_SETMASK：将当前的信号集合设置为信号阻塞集合
  - set: 当前进程的信号集
  - oset: 保存当前进程的信号阻塞集合
- 说明
  - 在使用之前要先设置好信号集合set

信号操作函数示例

❖ siginttest.c

```
/*自定义SIGINT的处理函数，如果你按ctrl+c，
则会打印提示，而不是默认的退出*/
void my_func(int sigo_num){
  printf("try 'ctrl+\\ to quit' .\n");
}

int main( ){
  sigset_t set;
  struct sigaction action1, action2;
  /*初始化信号集为空*/
  if (sigemptyset(&set) < 0) {
    perror("sigemptyset");
    exit(1);
  }
   /*将相应的信号加入信号集*/
  if (sigaddset(&set, SIGQUIT) < 0) {
    perror("sigaddset SIGQUIT");
    exit(1);
  }
  if (sigaddset(&set, SIGINT) < 0) {
     perror("sigaddset SIGINT");
     exit(1);
  }
```

信号操作函数示例

❖ **siginttest.c**

```
/*设置信号屏蔽字*/
 if (sigprocmask(SIG_BLOCK, &set, NULL) < 0) {
    perror("sigprocmask SIG_BLOCK");
    exit(1);
 } else {
    printf("blocked,and sleep for 5s ...\n");
    sleep(5);
 }
 if (sigprocmask(SIG_UNBLOCK, &set, NULL) < 0) {
    perror("sigprocmask SIG UNBLOCK");
    exit(1);
 } else {
    printf("unblock\n");
    /*此处可以添加函数功能模块process( )*/
    sleep(2);
    printf("If you want to quit this program, please try ...\n");
 }
```

信号操作函数示例

❖ siginttest.c

```
/*对相应的信号进行循环处理*/
while (1) {
   if (sigismember(&set, SIGINT)) {
      sigemptyset(&action1.sa_mask);
      action1.sa_handler = my_func;
      sigaction(SIGINT, &action1, NULL);
   }else
   if (sigismember(&set, SIGQUIT)) {
      sigemptyset(&action2.sa_mask);
      /*SIG_DFL采用缺省的方式处理*/
      action2.sa_handler = SIG_DFL;
      sigaction(SIGTERM, &action2, NULL);
   }
}
return 0;
}
```

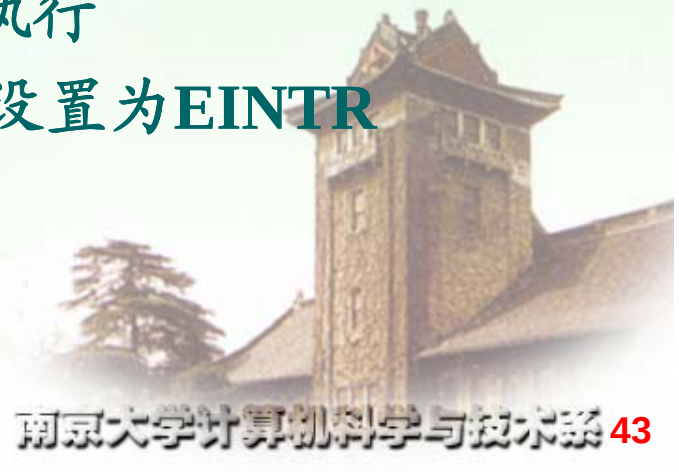
信号操作函数示例

- **siginttest.c**
  - 运行结果

```
localhost:~$./siginttest
blocked,and sleep for 5s ...
unblock
If you want to quit this program, please try ...
If you want to quit, please try 'ctrl+\' .
localhost:~$
```

其他信号处理函数

- **sigpending(sigset_t *set))**
  - 获得当前已递送到进程，但被阻塞的所有信号
  - 在set指向的信号集中返回结果
- **sigsuspend(const sigset_t *mask))**
  - 在接收到某个信号之前，临时用mask替换进程的信号掩码，并暂停进程执行，直到收到信号为止
  - sigsuspend 返回后将恢复调用之前的信号掩码
  - 信号处理函数完成后，进程将继续执行
  - 该系统调用始终返回-1，并将errno设置为EINTR



信号处理示例

```
int main(void){
   sigset_t set,pendset;
   struct sigaction action;
   //清空信号集
   sigemptyset(&set);
   //加入SIGTERM呢个信号
   sigaddset(&set,SIGTERM);
   //设为阻塞
   sigprocmask(SIG_BLOCK,&set,NULL);
   //因为阻塞所以就算用kill发送信号都无反应
   kill(getpid(),SIGTERM);
   //查下有什么阻塞信号,装入&pendset里面
   sigpending(&pendset);
```

信号处理示例

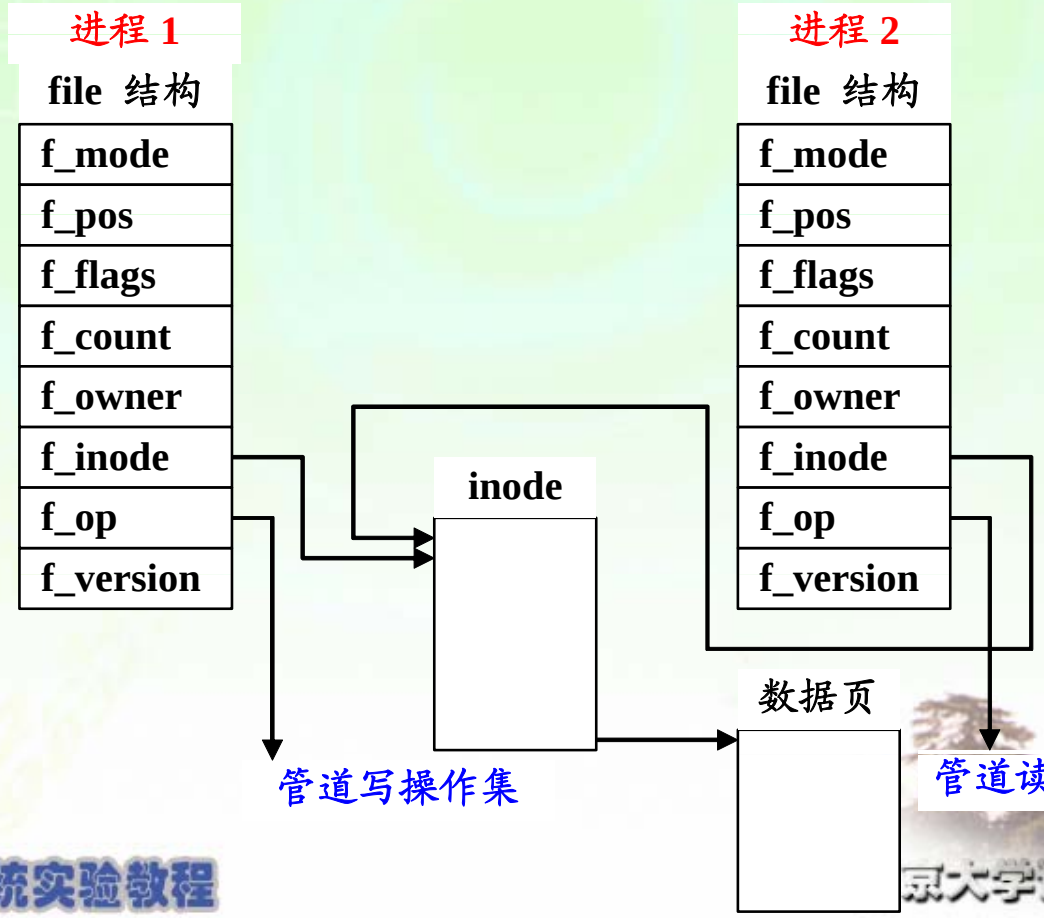
```
    //看下面有什么信号在里面
    if(sigismember(&pendset,SIGTERM)){
        printf("yes,the SIGTERM is here\n");
        //清空阻塞信号集
        sigemptyset(&action.sa_mask);
        //信号函数处理为ignore(忽略)
        action.sa_handler=SIG_IGN;
        //启动同signal功能类的信号函数
        sigaction(SIGTERM,&action,NULL);
    }
    //解除之前的信号阻塞
    sigprocmask(SIG_UNBLOCK,&set,NULL);
    exit(EXIT_SUCCESS);
}
```

主要内容

- 背景知识
  - 进程间的通信方式
  - 信号通信
  - 管道通信
- 实验内容
  - 信号通信
  - 匿名管道通信
  - 命名管道通信
  - 使用命名管道建立客户/服务器关联程序

Linux中的管道实现

- 实现原理
  - 将两个 file 结构指向同一个临时的 VFS 索引节点，而两个 VFS 索引节点又指向同一个物理页而实现管道



匿名管道

- 管道是所有Unix都提供的一种IPC机制
  - 对于管道两端的进程而言，是一个只存在于主存中的特殊文件
  - 管道是半双工的，数据只能向一个方向流动
    - ✓ 一个进程将数据写入管道，另一个进程从管道中读取数据
    - ✓ 写入的内容添加在管道缓冲区的末尾，每次都是从缓冲区头部读出数据
  - 双向通信的建立
    - ✓ 需要建立起两个管道
  - 数据的读出和写入
    - ✓ 写入的内容每次都添加在管道缓冲区的末尾，每次都是从缓冲区的头部读出数据
  - 使用限制
    - ✓ 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）

匿名管道的建立

- 基本函数
  - int pipe(int filedes[2]);
- 参数说明
  - filedes[2]描述管道两端
  - 管道两端功能固定
    - ✓ filedes[0]只能用于读，称为管道读端
    - ✓ filedes[1]只能用于写，称为管道写端
    - ✓ 若试图从写端读，或者向读端写都将导致错误发生
- 返回值
  - 成功时返回0
  - 失败时返回-1
- 说明
  - 一般文件的I/O函数都可用于管道，如close、read、write等

匿名管道的读写操作

❖ 读操作

- 如果管道的写端不存在，则认为已经读到数据的末尾，读函数返回的读出字节数为0。
- 当管道的写端存在时
  - ✓ 如果请求的字节数目大于PIPE_BUF，则返回管道中现有的数据字节数
  - ✓ 如果请求的字节数目不大于PIPE_BUF，则返回管道中现有数据字节数或者返回请求的字节数

❖ 写操作

- Linux将不保证写入的原子性
- 管道缓冲区一有空闲区域，写进程就会试图向管道写入数据
- 如果读进程不读出管道缓冲区中的数据，那么写操作将一直阻塞

❖ 说明

- 只有在管道的读端存在时，向管道中写入数据才有意义
- 否则，向管道中写入数据的进程将收到内核传来的SIGPIPE信号

匿名管道示例

- pipetest.c

```
#define MAX_LINE 80
int main( ){
  int thePipe[2], ret;
  char buf[MAX_LINE+1];
  const char *testbuf="a test string.";
  if ( pipe( thePipe ) == 0 ) {
    if (fork( ) == 0) {
      ret = read( thePipe[0], buf, MAX_LINE );
      buf[ret] = 0;
      printf( "Child read %s\n", buf );
    } else {
      ret = write( thePipe[1], testbuf, strlen(testbuf) );
      ret = wait( NULL );
    }
  }
  close(thePipe[0] );
  close(thePipe[1] );
  return 0;
}
```

有名管道

- 匿名管道缺点
  - 没有名字，只能用于具有亲缘关系的进程间通信
- FIFO，有名管道
  - 特殊的文件类型
    - ✓ 严格遵循先入先出的读写规则
    - ✓ 有名字，FIFO的名字包含在系统的目录树结构中，支持无亲缘关系的进程按名访问
    - ✓ 类似管道，在文件系统中不存在数据块，而是与一块内核缓冲区相关联

有名管道的建立

- 基本函数
  - int mkfifo(const char * pathname, mode_t mode);
- 参数说明
  - pathname：创建的FIFO名字
  - mode：规定FIFO的读写权限
- 返回值
  - 成功时返回0
  - 失败时返回-1
  - 若路径名存在，则返回EEXIST错误
- 说明
  - 一般文件的I/O函数都可用于管道，如open、 closeread、write等

有名管道的open()

- 打开规则
  - 打开操作是为读而打开FIFO
    - ✓若已经有相应进程为写而打开该FIFO，则当前打开操作将成功返回；
    - ✓否则，可能阻塞直到有相应进程为写而打开该FIFO（当前打开操作设置了阻塞标志）；
    - ✓或者，成功返回（当前打开操作没有设置阻塞标志）。
  - 打开操作是为写而打开FIFO
    - ✓如果已经有相应进程为读而打开该FIFO，则当前打开操作将成功返回；
    - ✓否则，可能阻塞直到有相应进程为读而打开该FIFO（当前打开操作设置了阻塞标志）；或者，返回ENXIO错误（当前打开操作没有设置阻塞标志）。

有名管道举例

❖ fifotest1.c(写管道)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define FIFO_SERVER "/tmp/fifoserver"
main(int argc,char** argv){
//参数为即将写入的字节数
  int fd;
  char w_buf[4096*2];
  int real_wnum;
  memset(w_buf,0,4096*2);
  if((mkfifo(FIFO_SERVER,O_CREAT|O_EXCL)<0)&&(errno!=EEXIST))
    printf("cannot create fifoserver\n");
  if(fd==-1)
    if(errno==ENXIO)
      printf("open error; no reading process\n");
  fd=open(FIFO_SERVER,O_WRONLY|O_NONBLOCK,0);
  //设置非阻塞标志
  //fd=open(FIFO_SERVER,O_WRONLY,0);
  //设置阻塞标志
  real_wnum=write(fd,w_buf,2048);
```

有名管道举例

❖ **fifotest1.c(写管道)**

```
  if(real_wnum==-1) {
    if(errno==EAGAIN)
      printf("write to fifo error; try later\n");
  }
  else
    printf("real write num is %d\n",real_wnum);
  real_wnum=write(fd,w_buf,5000);
  //5000用于测试写入字节大于4096时的非原子性
  //real_wnum=write(fd,w_buf,4096);
  //4096用于测试写入字节不大于4096时的原子性
  if(real_wnum==-1)
    if(errno==EAGAIN)
      printf("try later\n");
}
```

有名管道举例

❖ **fifotest2.c(读管道)**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define FIFO_SERVER "/tmp/fifoserver"
main(int argc,char** argv){
  char r_buf[4096*2];
  int  fd;
  int  r_size;
  int  ret_size;
  r_size=atoi(argv[1]);
  printf("requred real read bytes %d\n",r_size);
  memset(r_buf,0,sizeof(r_buf));
  fd=open(FIFO_SERVER,O_RDONLY|O_NONBLOCK,0);
  //fd=open(FIFO_SERVER,O_RDONLY,0);
  //在此处可以把读程序编译成两个不同版本：阻塞版本及非阻塞版本
  if(fd==-1)  {
    printf("open %s for read error\n");
    exit(-1);
  }
```

有名管道举例

❖ **fifotest2.c(读管道)**

```
  while(1)   {
    memset(r_buf,0,sizeof(r_buf));
    ret_size=read(fd,r_buf,r_size);
    if(ret_size==-1)
      if(errno==EAGAIN)
        printf("no data avlaible\n");
    printf("real read bytes %d\n",ret_size);
    sleep(1);
  }
  pause();
  unlink(FIFO_SERVER);
}
```

主要内容

- 背景知识
  - 进程间的通信方式
  - 信号通信
  - 管道通信
- 实验内容
  - 信号通信
  - 匿名管道通信
  - 命名管道通信
  - 使用命名管道建立客户/服务器关联程序

信号通信实验

❖实验说明

- 利用信号通信机制在父子进程、父子进程及兄弟进程间进行通信

❖解决方案

- 父子进程之间的通信可分为阻塞型通信和非阻塞型通信两种情形
- 父子进程之间的通信可分为阻塞型通信和非阻塞型通信两种情形
- 阻塞型通信情形
  - ✓父进程可以使用wait( )/waitpid( )等待子进程结束
  - ✓为避免僵死子进程的产生，也可以循环调用wait( )/watipid( )来等待所有子进程的结束
  - ✓但此时最好的方法是让子进程在结束时，向父进程发送的SIGCHLD信号，父进程通过signal( )/sigaction( )来响应子进程的结束

主要内容

- 背景知识
  - 进程间的通信方式
  - 信号通信
  - 管道通信
- 实验内容
  - 信号通信
  - 匿名管道通信
  - 命名管道通信
  - 使用命名管道建立客户/服务器关联程序

匿名管道通信实验

- 实验说明
  - 学习使用匿名管道在两进程间建立通信
- 解决方案
  - 匿名管道只能用于具有亲缘关系的进程间通信
    - 一个进程用pipe( )创建管道后，一般再fork( )一个子进程，然后通过管道实现父子进程间的通信，也可fork( )多个进程实现兄弟进程之间的通信
  - 管道两端分别用描述字fd[0]以及fd[1]来描述
    - 其中一端只能用于读，由描述字fd[0]表示，称其为管道读端
    - 另一端则只能用于写，由描述字fd[1]来表示，称其为管道写端

主要内容

- 背景知识
  - 进程间的通信方式
  - 信号通信
  - 管道通信
- 实验内容
  - 信号通信
  - 匿名管道通信
  - 命名管道通信
  - 使用命名管道建立客户/服务器关联程序

命名管道通信实验

- 实验说明
  - 学会使用有名管道在多进程间建立通信
- 解决方案
  - 有名管道以FIFO的文件形式存在于文件系统中
  - 只要可以访问该文件路径，就能够彼此通过有名管道相互通信

主要内容

- 背景知识
  - 进程间的通信方式
  - 信号通信
  - 管道通信
- 实验内容
  - 信号通信
  - 匿名管道通信
  - 命名管道通信
  - 使用命名管道建立客户/服务器关联程序

使用命名管道建立客户/服务器关联程序实验

❖实验说明

- 使用有名管道建立一个客户与服务器程序的关联，以便实现数据共享

❖解决方案

- 使用有名管道建立一个客户与服务器程序的关联，以便实现数据共享
- 为支持客户与服务器程序共享数据，程序首先在两端建立一个有名管道随后一方可通过该有名管道写入数据，另一方可通过该管道获取数据
- 程序设计的关键在于，协调好两端的读写过程，避免同时读或写
- 具体而言，可在程序中设置标记符来标识共享数据的结束

第3章 传统的进程间通信