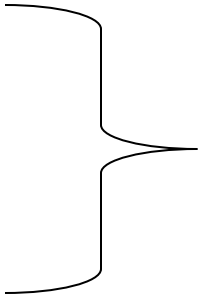
实验9 传统进程间通信

- 信号通信
- 管道通信
- 消息队列
- 信号量通信
- 共享主存通信
- 套接字



System V进程间通信

信号

- 又称软中断
  - 在软件层次上对中断机制的一种模拟
- 异步通信机制
- 信号源
  - 硬件
    - 用户按下Ctrl+C
    - 其它硬件故障
  - 软件
    - 信号发送函数
      - 用于在进程之间传递软中断信号
      - kill(), raise(), alarm(), setitimer(), sigqueue()

进程接收到信号后的处理方法

- 忽略
  - 忽略接收到的信号，不做任何处理
  - SIGKILL和SIGSTOP信号不能被忽略
- 捕获信号
  - 进程为需要处理的信号定义信号处理函数
  - 当信号发生时，执行对应的信号处理函数
  - 类似中断处理程序
- 默认操作
  - 由内核的默认处理程序处理

信号的存储

- 进程表项中的信号域，每一位对应一个信号
- sigpending结构用于维护本进程中的未决信号

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

  - signal是所有未决信号集
  - sigqueue刻画一个特定信号所携带的信息

信号类型

- kill –l命令可以获取Linux支持的信号列表
- 分类
  - 可靠 vs 不可靠
  - 实时 vs 非实时

信号名称	说明
SIGABORT	进程异常终止
SIGALRM	超时警告
SIGFPE	浮点运算异常
SIGHUP	连接挂起
SIGILL	非法指令
SIGINT	终端中断
SIGKILL	终止进程
SIGPIPE	向无读进程的管道写数据
SIGQUIT	终端退出
SIGSEGV	无效内存段访问
SIGTERM	终止
SIGUSR1	用户定义信号1
SIGUSR2	用户定义信号2

信号名称	说明
SIGCHLD	子进程已经停止或退出
SIGCONT	继续执行暂停进程
SIGSTOP	停止执行
SIGTSTP	终端挂起
SIGTTIN	后台进程尝试读操作
SIGTTOU	后台进程尝试写操作

信号

- Ctrl+C组合键会向前台进程(当前正在运行的程序)发送SIGINT信号
- 若要给非前台进程发信号，需要用kill命令
  - kill –HUP 512

信号生命周期

- 信号诞生
  - 触发信号的事件发生
  - 如，硬件异常，定时器超时，调用信号发送函数等
- 信号在进程中注册
  - 将信号值加入到进程的未决信号集中
  - 进程知道信号的存在，但尚未来得及处理，或被进程阻塞
  - 实时信号每次都注册，相同的实时信号可能占据多个sigqueue结构
  - 非实时信号则至多占用一个sigqueue结构
- 信号在进程中注销
  - 进程执行过程中会检测是否有信号等待处理
  - 若存在未决信号等待处理且该信号未被进程阻塞，则运行相应的信号处理函数前，将信号在未决信号链中占有的结构删除
  - 当实时信号的所有相同信号都从未决信号链sigqueue中删除后，同时将信号从未决信号集signal中删除
- 信号处理函数执行完毕
  - 进程注销信号后，执行相应的信号处理函数
  - 执行完毕后，信号的本次发送对进程的影响彻底结束

信号处理函数

- 信号安装函数
- 信号发送函数
- 信号集操作函数

信号安装函数

- 用于设置某个信号的处理函数
  - signal()
  - sigaction()

  需要包含#include <signal.h>头文件

signal()

- signal函数原型
  - void (*signal(int sig, void (*func)(int)))(int);
- 说明
  - signal函数包含两个参数
    - 准备捕获或忽略的信号由sig参数给出
    - 接收到指定信号后将要调用的函数由func给出
      - func必须是一个接收整型参数，返回类型为void的函数
  - signal的返回值是与func同类型的函数，即先前用来处理这个信号的函数
  - 可以用下述两个特殊值之一来代替信号处理函数
    - SIG_IGN：忽略信号
    - SIG_DFL：恢复默认行为

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

void ouch(int sig){
    printf("OUCH! –I got signal %d\n", sig);
    signal(SIGINT, SIG_DFL);   //尝试注释该语句，查看运行结果
}

int main(){
    signal(SIGINT, ouch);
     while(1){
         printf("Hello World!\n");
         sleep(1);
     }
 }
```

```
/*sigtest.c*/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void sigroutine(int sig){
    switch(sig){
        case 1:
            printf("Get a signal –SIGHUP\n");
            break;
        case 2:
            printf("Get a signal –SIGINT\n");
            break;
         case 3:
            printf("Get a signal –SIGQUIT\n");
            break;
      }
     return;
}

int main(){
      printf("Process id is %d\n", getpid());
      signal(SIGHUP, sigroutine);
      signal(SIGINT, sigroutine);
      signal(SIGQUIT, sigroutine);
      for(; ;);
}
```

$./sigtest

Process id is 13657

Get a signal –SIGINT /*按下Ctrl+C*/

Get a signal –SIGQUIT /*按下Ctrl+\ */

[3]+已停止 ./sigtest /*按下Ctrl+Z*/

$ kill –SIGKILL 13657 (进程号使用前面的输出进程号)

$./sigtest & (后台运行)

[4] 13661

$ Process id is 13661

kill –HUP 13661

Get a signal –SIGHUP

$kill –SIGINT 13661

Get a signal –SIGINT

$kill –SIGKILL 13661

$kill –SIGHUP 13661

Bash: kill: (13661)-没有那个进程

[4] 已杀死 ./sigtest

Sigaction()

- 函数原型

```
int sigaction(int sig, const struct sigaction *act, struct
  sigaction *oact);
struct sigaction{
    void (*sa_handler)(int sig);
    void (*sa_sigaction)(int siginfo_t *info, void *act);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restore)(void);
}
```

Sigaction()

- sigaction函数说明
  - 三个参数
    - 要设置处理函数的信号由第一个参数sig给出
    - act包含对该信号进行处理的信息
    - oact用于保存以前对这个信号的处理信息
- sigaction结构说明
  - sa_handler是函数指针，指向与信号处理对应的处理函数
  - sa_sigaction, sa_restore与sa_handler功能类似，但参数不同
  - sa_flags用来设置各种信号操作，一般设为0
  - sa_mask用于设置sa_hander中屏蔽的信号集，即在调用sa_hanlder之前，将sa_mask中给出的信号集加入进程的信号屏蔽字中

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void ouch(int sig){
    printf("OUCH! I got signal %d\n", sig);
}
int main(){
    struct sigaction act;
    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);
    while(1){
        printf("Hello World!\n");
        sleep(1);
    }
}
```

信号发送函数

- int kill(pid_t pid, int sig);
  - 把参数sig给定的信号发送给由参数pid给出的进程号所指向的进程
  - 必须拥有相应的权限（如两个进程拥有相同的用户ID）
  - 超级用户可以发信号给任何进程
- unsigned int alarm(unsigned int seconds);
  - 在seconds秒后向进程发送一个SIGALRM信号（只发送一次）
  - 参数为0，则之前设置的闹钟将会被取消
  - 返回值返回之前闹钟的剩余秒数，如果之前未设闹钟则返回0

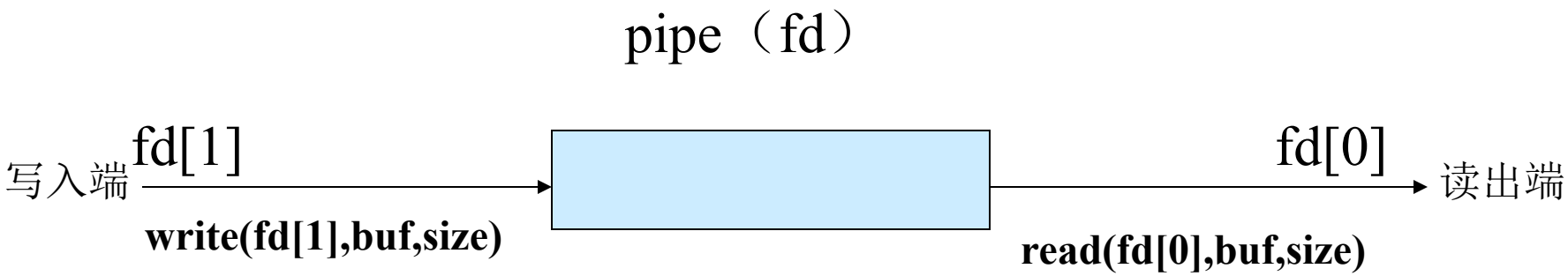
```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

static int alarm_fired=0;
void ding(int sig){
    alarm_fired=1;
}

int main(){
    pid_t pid;
    printf("alarm application starting\n");
    pid = fork();
    switch(pid){
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            sleep(5);
            kill(getppid(), SIGALRM);        /*向父进程发送SIGALRM信号*/
            exit(0);
     }
    printf("waiting for alarm to go off\n");
    signal(SIGALRM, ding);              /*父进程注册信号SIGALRM */
    pause();                             /*挂起程序的执行直到有一个信号出现为止*/
    if(alarm_fired)
        printf("Ding!\n");
    printf("done\n");
   exit(0);
}
```

管道通信

- 匿名管道
- 命名管道

匿名管道

- 只能用于具有亲缘关系的进程间通信，如父子进程或兄弟进程；
- 半双工，数据只能单向流动
- 匿名管道对于通信两端的进程而言，就是一个文件，可以使用一般的文件I/O函数，如read(), write();
- 一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区末尾，每次都从缓冲区的头部读出数据。
- 语法：int pipe(int fd[2]);
  - 返回值0，表示成功
  - -1， 表示失败



管道按先进先出（FIFO）方式F传送消息，且只能单向传送消息

管道文件的读写操作的同步与互斥

- 对管道文件进行读写操作过程中要对发送进程和接送进程实施正确的同步与互斥以确保通信的正确性：
  - 接收进程：当接收进程读pipe时，若发现pipe为空，则进入等待状态。一旦有发送进程对该pipe执行写操作时唤醒等待进程.
  - 发送进程：当发送进程在写pipe时，总是先按pipe文件的当前长度设置，如果pipe文件长度已经到规定长度，但仍有一部分信息没有写入，则系统使要求写pipe的进程进入睡眠状态，当读pipe进程读走了全部信息时，系统再唤醒待写的进程。它将余下部分信息继续送入pipe中。

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <wait.h>
#define MAX_LINE 180
Int main()
{
   int thePipe[2], ret;
   char buf[MAX_LINE+1];
   const char *testbuf = “a test string”;
   if(pipe(thePipe)==0){                    /*创建匿名管道*/
      if(fork()==0){                         /*子进程*/
         ret = read(thePipe[0], buf, MAX_LINE);    /*从管道中读数据*/
         buf[ret]=0;
         printf(“Child read %s\n”, buf);
      }
      else{                                          /*父进程*/
         sleep(5);
         ret = write(thePipe[1], testbuf, strlen(testbuf));   /*向管道写入数据*/
         ret = wait(NULL);                                  /*等待子进程结束*/
      }
      close(thePipe[0]);
      close(thePipe[1]);
    }
  return 0;
}
```

Linux 匿名管道进程通信例子

命名管道

- 可用于任何可以访问命名文件的两个进程之间的通信
- 创建：
  - int mkfifo(const char *filename, mode_t mode);
  - mode与文件创建系统调用open()中的mode含义一致
- 打开管道：open()
- 读管道：read()
- 写管道：write()
- 命名管道是先进先出的，不支持lseek等文件定位操作

```
/*fifo_h.h*/
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
 #include <fcntl.h>
#include <string.h>
#define FIFO_PATH “/home/stu/fifo_abc”
struct fifo_cmd {
    pid_t pid;
    char cmd[100];
};
```

Linux 命名通信管道例子

```
/*server.c*/
#include “fifo_h.h”
int main(void) {
    int fd,err,n;
    struct fifo_cmd cmd;
    if((err = mkfifo(FIFO_PATH, 0666)) < 0){
       if(err != EEXIST) {
          perror("mkfifo fail");
          exit(-1);
       }
    }
   if((fd = open(FIFO_PATH, O_RDONLY)) < 0){
       perror("open fail");
       exit(-1);
    }
   while(1) {
       if((n = read(fd, &cmd, sizeof(cmd))) < 0) {
           perror("read fail");
           exit(-1);
       }
       if(n > 0) {
           printf("command from process %d: %s\n", cmd.pid, cmd.cmd);
       }
      sleep(1);
   }
 }
```

```
/*client.c*/
#include “fifo_h.h”
int main(int argc, char* argv[]) {
    int fd;
    struct fifo_cmd cmd;
    if((fd = open(FIFO_PATH, O_WRONLY)) < 0) {
      perror("open fail");
      exit(-1);
     }
     cmd.pid = getpid();
     while(1) {
        printf("Please enter a command string: ");
        fgets(cmd.cmd, sizeof(cmd.cmd), stdin);
        cmd.cmd[strlen(cmd.cmd) - 1] = 0;
        if(write(fd, &cmd, sizeof(cmd)) < 0) {
           perror("write fail");
           exit(-1);
         }
     }
}
```

Linux 命名通信管道例子

System V消息队列

- 创建消息队列
  - int msgget(key_t key, int flag);
  - 若调用成功，返回消息队列标识
  - flag指定创建/打开方式，如IPC_CREAT, IPC_EXCL, IPC_NOWAIT
- 向消息队列发送消息
  - int msgsnd(int msgid, struct msgbuf *msgp, size_t size, int flag);
  - 参数说明
    - msgid为msgget返回的消息队列标识
    - msgp为指向消息缓冲区的指针，用于暂时存储发送和接收的消息
    - size指消息的大小
    - flag一般设为0
- 从消息队列接收一个消息
  - ssize_t msgrcv(int msgid, void *msgp, size_t size, long msgtype, int flag);
    - msgtype指消息类型
- 在消息队列上执行指定的操作
  - int msgctl(int msgid, int cmd, struct msgid_ds *buf);

System V消息队列

- 在System V的IPC机制(消息队列、共享内存、信号量)实现中，需要用ftok函数创建一个ket_t类型的值，并作为相应资源的键值；
- key_t ftok(char *fname, int id);
- 一般的UNIX实现中，是将文件的索引节点号取出，前面加上子序号id得到key_t的返回值。
  - ls –i 命令可以查看一个节点的inode号

```
/*msgqueue.h*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <error.h>
#define BUFFER 10
#define MSG_PATH "/home/stu"
#define PRIVATE_KEY 10001
struct msgbuf{
  long mtype;
  char buf[BUFFER];
};
```

//每个消息都用类似的数据结构来表示

System V消息队列实例

```
/*sender.c*/
#include "msgqueue.h"
int main(){
  key_t msgkey;
  int msgid;
  struct msgbuf msg;
  msgkey = ftok(MSG_PATH, PRIVATE_KEY);
  if(msgkey == -1) {  perror("ftok error");  exit(-1); }
  msgid = msgget(msgkey, IPC_CREAT|0666);
  if(msgid == -1) {  perror("msgget error");  exit(-1); }
  int i = 0;
  while(i<10){
    memset(msg.buf, 0, BUFFER);
    sprintf(msg.buf, "buf_0x%x", i);
    msg.mtype=1001;
    if( msgsnd(msgid, &msg, sizeof(struct msgbuf), 0) <0)  { perror("msgsnd"); exit(-
      1);  }
    i++;
    sleep(1);
  }
  sleep(30);
  if( msgctl(msgid, IPC_RMID, 0) ==-1){ perror("msgctl error"); exit(-1); }
  return 0;
}
```

System V消息队列实例

```
/*receiver.c*/
#include "msgqueue.h“
struct msgbuf msg;
int main(){
  key_t msgkey;
  int msgid;
msgkey = ftok(MSG_PATH, PRIVATE_KEY);
  if(msgkey == -1){  perror("ftok error");  exit(-1); }
  msgid = msgget(msgkey, IPC_EXCL|0666) ;
  if(msgid==-1){  perror("msgget error");  exit(-1); }
  int i = 0;
  while(i<10){
    memset(msg.buf, 0, BUFFER);
    msg.mtype=1001;
    if( msgrcv(msgid, &msg, sizeof(struct msgbuf), msg.mtype, 0) == -1){
      perror("msgrcv error");
      exit(-1);
    }
    printf("msg.buf = %s\n", msg.buf);
    i++;
    sleep(2);
  }
  return 0;
}
```

System V消息队列实例

System V共享内存API实例

- shmget()创建共享内存
- shmat()将共享内存连接到进程
- shmdt()将共享内存与进程解除连接
- shmctl()删除共享内存

```
/*shmtest.h*/
#include <sys/ipc.h>
#include <sys/shm.h>
 #include <sys/types.h>
 #include <unistd.h>
typedef struct{
    char name[4];
    int age;
} people;
```

```
/*testwrite.c*/
#include “shmtest.h”
void main(int argc, char** argv) {
    int shm_id,i;
    key_t key;
    char temp;
    people *p_map;
    char* name = "/dev/shm/myshm2";
    key = ftok(name,’a’);
    if(key==-1) {
       perror("ftok error");
       return;
    }
    shm_id=shmget(key, 4096, IPC_CREAT|0666);   //create the shared memory;
    if(shm_id==-1) {
       perror("shmget error");
       return;
    }
    p_map=(people*)shmat(shm_id,NULL,0);   //attach it to this process
    if(p_map == (void *) -1){
       perror(“shmat error\n”);
       return;
    }
    temp='a';
    for(i = 0;i<10;i++) {
       temp+=1;
       memcpy((*(p_map+i)).name,&temp,1);
        (*(p_map+i)).age=20+i;
     }
   if( shmdt(p_map) ==-1)     //detach the shared memory
      perror(" detach error ");
}
```

```
/*testread.c*/
#include “shmtest.h”
void main(int argc, char** argv) {
    int shm_id,i;
    key_t key;
    people *p_map;
    char* name = "/dev/shm/myshm2";
    key = ftok(name,’a’);
    if(key == -1) {
        perror("ftok error");
        return;
    }
    shm_id = shmget(key,4096,IPC_CREAT);  //create the shared memory;
    if(shm_id == -1) {
        perror("shmget error");
        return;
     }
    p_map = (people*)shmat(shm_id,NULL,0);
     for(i = 0;i<10;i++) {
        printf( "name:%s\n",(*(p_map+i)).name );
        printf( "age %d\n",(*(p_map+i)).age );
     }
      if(shmdt(p_map) == -1)
          perror(" detach error ");
      if( shmctl(shm_id, IPC_RMID, 0) == -1)
          printf(“shmctl(IPC_RMID) failed\n”);
}
```