

Linux操作系统

进程通信

主讲：杨东平
中国矿大计算机学院

进程通信概述

进程通信的概念

- ✓ 进程用户空间是相互独立的，一般而言是不能相互访问的
- ✓ 但很多情况下进程间需要互相通信，来完成系统的某项功能
- ✓ 进程通过与内核及其它进程之间的互相通信来协调它们的行为

进程通信的应用场景

- ✓ **数据传输**：一个进程需要将它的数据发送给另一个进程
- ✓ **共享数据**：多个进程操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到
- ✓ **通知事件**：一个进程需要向另一个或一组进程发送消息，通知它(它们)发生了某种事件(如进程终止时要通知父进程)
- ✓ **资源共享**：多个进程之间共享同样的资源。为此，需要内核提供锁和同步机制
- ✓ **进程控制**：有些进程希望完全控制另一个进程的(如 Debug 进程)，此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变

网络安全与网络工程系杨东平 jsxhbc@163.com Linux操作系统 2020年3月2日 4时50分 2

进程通信的方式

- Ø 管道(pipe)
- Ø 信号量(semaphore)
- Ø 消息队列(message queue)
- Ø 信号 (signal)
- Ø 共享内存(shared memory)
- Ø 套接字(socket)

网络安全与网络工程系杨东平 jsxhbc@163.com Linux操作系统 2020年3月2日 4时50分 3

进程通信：管道(pipe)

Ø 管道是由内核管理的一个缓冲区，它被设计成为环形的数据结构

- ✓ 管道的两端分别连接连个通信的进程，当两个进程都终结的时候，管道也自动消失

Ø 管道包括三种：

- ✓ 普通管道(无名管道) pipe：通常有两种限制
 - F 单工
 - F 只能在父子或兄弟进程间使用
- ✓ 流管道 s_pipe
 - F 可半双工通信
 - F 只能在父子或兄弟进程间使用
- ✓ 命名管道 named_pipe
 - F 可以在不相关的进程之间进行半双工通讯

网络安全与网络工程系杨东平 jsxhbc@163.com Linux操作系统 2020年3月2日 4时50分 4

管道数据的读写

Ø 将数据写入管道：write() 函数

- ✓ write() 函数的使用与写文件操作相同，但需要注意的是管道的长度受到限制，管道满时写入操作将会被阻塞

Ø 从管道中读取数据：read() 函数

- ✓ read() 函数的使用与读文件操作相同，读取的顺序与写入顺序相同。当数据被读取后，这些数据将自动被管道清除
- ✓ 如果管道为空，并且管道写入端口是打开的，则 read() 函数将被阻塞

网络安全与网络工程系杨东平 jsxhbc@163.com Linux操作系统 2020年3月2日 4时50分 5

无名管道

Ø 头文件：unistd.h

Ø 创建管道原型：int pipe(int filedes[2]);

- ✓ 参数 filedes 返回两个文件描述符：

- F filedes[0] 用于读出数据，读取时必须关闭写入端，即 close(filedes[1]);
- F filedes[1] 用于写入数据，写入时必须关闭读取端，即 close(filedes[0]);

网络安全与网络工程系杨东平 jsxhbc@163.com Linux操作系统 2020年3月2日 4时50分 6

例：子进程用无名管道向父进程传递数据(视频：43 进程通信：无名管道)

```
#include <stdio.h> [root@localhost ~]# gcc -o unnamedpipe unnamedpipe.c
#include <unistd.h> [root@localhost ~]# ./unnamedpipe
#include <stdlib.h> in the spawning(parent) process...
#include <string.h> in the spawned(child) process...
void main()
{
    int file_descriptors[2];
    pid_t pid;
    char buf[256];
    int returned_count;
    pipe(file_descriptors);
    if((pid = fork()) == -1) {
        printf("Error in forkin");
        exit(1);
    } else if(pid == 0) {
        printf("in the spawned (child) process...\n");
        close(file_descriptors[0]);
        write(file_descriptors[1], "test data", strlen("test data"));
        exit(0);
    } else {
        printf("in the spawning (parent) process...\n");
        close(file_descriptors[1]);
        returned_count = read(file_descriptors[0], buf, sizeof(buf));
        printf("%d bytes of data received from spawned process: %s\n", returned_count, buf);
    }
}
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 7

命名管道

Linux 提供了 FIFO 方式连接进程，FIFO 又称为命名管道

头文件：
sys/types.h
sys/stat.h

函数原型：`int mkfifo(const char *filename, mode_t mode);`

✓ FIFO 在文件系统中表现为一个文件，大部分的系统文件调用都可以用在 FIFO 上面，如：read、open、write、close、unlink、stat 等，但 seek 等函数不能对 FIFO 调用

✓ filename 是有名管道的路径，包含了有名管道文件的名字，如："/tmp/myfifo"

✓ mode 是对管道的读写权限，是个八进制数，如 0777

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 8

命名管道(续)

用 open 函数打开命名管道时需要注意两点：

- ✓ 1) 不能以 O_RDWR 模式打开命名管道 FIFO 文件，否则其行为是未定义的，管道是单向的，不能同时读写
- ✓ 2) 传递给 open 调用的是 FIFO 的路径名，而不是正常的文件

打开 FIFO 文件通常有四种方式：

- ✓ 只读、阻塞模式：`open(pathname, O_RDONLY);`
- ✓ 只读、非阻塞模式：`open(pathname, O_RDONLY | O_NONBLOCK);`
- ✓ 只写、阻塞模式：`open(pathname, O_WRONLY);`
- ✓ 只写、非阻塞模式：`open(pathname, O_WRONLY | O_NONBLOCK);`

阻塞模式 open 打开 FIFO：

- ✓ 1) 当以阻塞、只读模式打开 FIFO 文件时，将会阻塞直到其他进程以写方式打开访问文件
- ✓ 2) 当以阻塞、只写模式打开 FIFO 文件时，将会阻塞直到其他进程以读方式打开文件
- ✓ 3) 当以非阻塞方式(指定 O_NONBLOCK)方式只读打开 FIFO 的时候，则立即返回。当只写打开时，如果没有进程为读打开 FIFO，则返回 -1，其 errno 是 ENXIO

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 9

例：命名管道通信(阻塞式：发送数据: namedpipe_write.c)

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

void main()
{
    char *buf = "I am write process\n";
    int fd = open("my_fifo", O_WRONLY); // my_fifo 管道路径，必须与读者相同
    write(fd, buf, strlen(buf));
    close(fd);
}

Last login: Fri Sep 28 13:31:06 2018 from 192.168.116.1
[root@localhost ~]# ./namedpipe_write
[root@localhost ~]#
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 10

例：命名管道通信(阻塞式：接收数据: namedpipe_read.c)

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

void main()
{
    int ret = mkfifo("my_fifo", 0777); // my_fifo 管道路径，必须与写者相同
    if (ret == -1)
    {
        printf("make fifo failed\n");
        return;
    }

    char buf[256] = {0};
    int fd = open("my_fifo", O_RDONLY);
    read(fd, buf, 256);
    printf("%s\n", buf);
    close(fd);
    unlink("my_fifo");
}
```

源代码及编译视频：44 进程通信：命名管道的源代码

通信过程(视频：45 进程通信：命名管道通信)

F 1) 因接收程序(namedpipe_read.c) 中创建命名管道，而发送程序则是使用已创建的管道 (namedpipe_write.c)，故必须先运行接收程序，再运行发送程序

F 2) 两个程序分别在不同的终端中运行

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 11

进程通信：信号量(semaphore)

信号量的工作原理

✓ 信号量只能进行等待和发送信号两种操作

P(sv)：

sv>0 时，sv=sv-1

sv=0 时，挂起进程的执行

V(sv)：

如果有其他进程因等待 sv 而被挂起，就让它恢复运行

如果没有进程因等待 sv 而挂起，就给它加 1

✓ 主要作为进程间以及同一进程不同线程之间的同步手段

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 12


```
int main(int argc, char *argv[])
{
    char message = 'X'; // 屏幕显示的信息
    int i = 0;           // 循环控制变量

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT); // 创建信号量

    if(argc > 1)
    {
        if(!set_semvalue()) // 程序第一次被调用, 初始化信号量
        {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        message = argv[1][0]; // 设置要输出到屏幕中的信息, 即其参数的第一个字符
        sleep(2);
    }
}
```

```
for(i = 0; i < 10; ++i)
{
    if(!semaphore_p()) // 进入临界区
    {
        exit(EXIT_FAILURE);
    }

    printf("%c", message); // 向屏幕中输出数据
    fflush(stdout);         // 清理缓冲区, 然后休眠随机时间
    sleep(rand() % 3);
    printf("%c", message); // 离开临界区前再一次向屏幕输出数据
    fflush(stdout);
    if(!semaphore_v())     // 离开临界区, 休眠随机时间后继续循环
    {
        exit(EXIT_FAILURE);
    }
    sleep(rand() % 2);
}
sleep(10);
printf("\n%d - finished\n", getpid());
```

```
if(argc > 1)
{
    sleep(3); // 如果程序是第一次被调用, 则在退出前删除信号量
    del_semvalue();
}
exit(EXIT_SUCCESS);

static int set_semvalue()
{
    union semun sem_union; // 用于初始化信号量, 在使用信号量前必须这样做

    sem_union.val = 1;
    if(semctl(sem_id, 0, SETVAL, sem_union) == -1)
    {
        return 0;
    }
    return 1;
}
```

```
static void del_semvalue() // 删除信号量
{
    union semun sem_union;
    if(semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
    {
        fprintf(stderr, "Failed to delete semaphore\n");
    }
}

static int semaphore_p() // 对信号量做减1操作, 即等待P(sv)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = -1; // P操作
    sem_b.sem_flg = SEM_UNDO;
    if(semop(sem_id, &sem_b, 1) == -1)
    {
        fprintf(stderr, "semaphore_p failed\n");
        return 0;
    }
    return 1;
}
```

```
static int semaphore_v() // 释放操作, 它使信号量变为可用, 即发送信号V(sv)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = 1; // V操作
    sem_b.sem_flg = SEM_UNDO;
    if(semop(sem_id, &sem_b, 1) == -1)
    {
        fprintf(stderr, "semaphore_v failed\n");
        return 0;
    }
    return 1;
}
```

```
[root@localhost ~]# ./semaphore1 A
AAAAAAAAAAAAAAAAAAAAAAAAA
1979 - finished
```

视频: [46 进程通信: 信号量 semaphore1.c](#)

示例: 信号量集合

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<errno.h>
#include<string.h>
#include<stdlib.h>
#include<assert.h>
#include<time.h>
#include<unistd.h>
#include<sys/wait.h>
```

```
#define MAX_SEMAPHORE 10 // 信号量集合中的信号数量
#define FILE_NAME "semaphore2.c" // 将取得值的文件名
```

```
union semun{
    int val;
    struct semid_ds* buf;
    unsigned short* array;
    struct seminfo* _buf;
}arg;
```

```

struct semid_ds sembuf;

int main()
{
    key_t key;
    int semid, ret, i;
    unsigned short buf[MAX_SEMAPHORE];
    struct sembuf sb[MAX_SEMAPHORE];
    pid_t pid;

    pid = fork();
    if(pid < 0) // 创建进程失败
    {
        fprintf(stderr, "Create Process Error!:%s\n", strerror(errno)); // strerror()取得错误的消息串
        exit(1);
    }
}

```

```

else if(pid > 0) // 在父进程中
{
    key = ftok(FILE_NAME, 'a'); // 根据文件或目录名及指定的数字，为IPC对象生成一个唯一的键值
    if(key == -1) {
        fprintf(stderr, "Error in ftok:%s!\n", strerror(errno));
        exit(1);
    }

    semid = semget(key, MAX_SEMAPHORE, IPC_CREAT | 0666); // 创建信号量集合
    if(semid == -1) {
        fprintf(stderr, "Error in semget:%s!\n", strerror(errno));
        exit(1);
    }
    printf("Semaphore have been initialed successfully in parent process, ID is :%d\n", semid);
    sleep(2);
    printf("parent wake up...\n");
    // 父进程在子进程等待 semaphore 的时候请求 semaphore，此时父进程将阻塞直至子进程释放 semaphore
    // 此时父进程的阻塞是因为 semaphore 不能申请，因而导致的进程阻塞
    for(i=0; i<MAX_SEMAPHORE; ++i)
    {
        sb[i].sem_num = i;
        sb[i].sem_op = -1; // 表示申请 semaphore
        sb[i].sem_flg = 0;
    }
}

```

```

    printf("parent is asking for resource...\n");
    ret = semop(semid, sb, 10); // p1 操作
    if(ret == 0) {
        printf("parent got the resource!\n");
    }
    waitpid(pid, NULL, 0); // 父进程等待子进程退出
    printf("parent exiting .. \n");
    exit(0);
}
else // 在子进程中
{
    key = ftok(FILE_NAME, 'a');
    if(key == -1) {
        fprintf(stderr, "Error in ftok:%s!\n", strerror(errno));
        exit(1);
    }

    semid = semget(key, MAX_SEMAPHORE, IPC_CREAT | 0666);
    if(semid == -1) {
        fprintf(stderr, "Error in semget:%s!\n", strerror(errno));
        exit(1);
    }
    printf("Semaphore have been initialed successfully in child process, ID is:%d\n", semid);
}

```

```

for(i=0; i<MAX_SEMAPHORE; ++i) // 初始化信号量
{
    buf[i] = i + 1;
}

arg.array = buf;
ret = semctl(semid, 0, SETALL, arg);
if(ret == -1)
{
    fprintf(stderr, "Error in semctl in child:%s!\n", strerror(errno));
    exit(1);
}
printf("In child , Semaphore Initialed!\n");

// 子进程在初始化了 semaphore 之后，就申请获得 semaphore
for(i=0; i<MAX_SEMAPHORE; ++i)
{
    sb[i].sem_num = i;
    sb[i].sem_op = -1;
    sb[i].sem_flg = 0;
}

```

```

ret = semop(semid, sb, 10); // 信号量0被阻塞
if(ret == -1)
{
    fprintf(stderr, "Child got semaphore failed: %s\n", strerror(errno));
    exit(1);
}

printf("child got semaphore, and start to sleep 3 seconds!\n");
sleep(3);
printf("child wake up.\n");
for(i=0; i<MAX_SEMAPHORE; ++i)
{
    sb[i].sem_num = i;
    sb[i].sem_op = 1;
    sb[i].sem_flg = 0;
}

printf("child start to release the resource...\n");
ret = semop(semid, sb, 10); // v操作
if(ret == -1)
{
    fprintf(stderr, "Child release semaphore failed:%s\n", strerror(errno));
    exit(1);
}
}


```

```

ret = semctl(semid, 0, IPC_RMID);
if(ret == -1)
{
    fprintf(stderr, "delete semaphore failed:%s! \n", strerror(errno));
    exit(1);
}

printf("child exiting successfully!\n");
exit(0);
}
return 0;
}

```



```

$ gcc 12-4.c
$ ./12-4
parent is asking for resource...
parent got the resource!
parent exiting ..
child got semaphore, and start to sleep 3 seconds!
child wake up.
child start to release the resource...
child exiting successfully!
delete semaphore failed: No such semaphore: 1
child exiting successfully!

```

进程通信：消息队列(message queue)

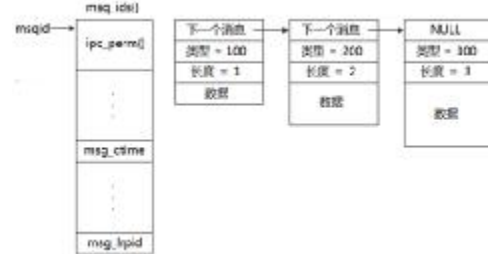
消息队列(message queue)：

- 消息队列是由内核管理的内部链表，用于进程之间传递信息
- 它克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点
- 消息队列是通过 IPC 标识符来区别的，不同的消息队列之间是相互独立的链表。
- 有足够权限的进程可以顺序地发送消息到消息队列中
- 被赋予读权限的进程可以读走消息队列中的消息

消息队列进行通信的进程可以是不相关的进程

使用消息队列的头文件

- sys/types.h
- sys/ipc.h
- sys/msg.h



消息队列中的数据结构

msqid_ds 内核数据结构

- Linux 内核维护每个消息队列的结构体，它保存着消息队列当前状态信息

```
struct msqid_ds{
    struct ipc_perm msg_perm; //所有者和权限
    time_t msg_stime; //最后一次调用 msgsnd 的时间
    time_t msg_rtime; //最后一次调用 msgrcv 的时间
    time_t msg_ctime; //队列最后一次变动的时间
    unsigned long __msg_cbytes; //当前队列中字节数(不标准)
    msgqnum_t msg_qnum; //当前队列中消息数
    msglen_t msg_qbytes; //队列中允许的最大字节数
    pid_t msg_lspid; //最后一次调用 msgsnd 的 PID
    pid_t msg_lrpid; //最后一次调用 msgrcv 的 PID
};
```

消息队列中的数据结构(续)

- Linux 内核在结构体 ipc_perm 中保存消息队列的一些重要的信息，如消息队列关联的键值、消息队列的用户 id、组 id 等

```
struct ipc_perm{
    key_t key; //消息队列键值
    uid_t uid; //有效的拥有者 UID
    gid_t gid; //有效的拥有者 GID
    uid_t cuid; //有效的创建者 UID
    gid_t cgid; //有效的创建者 GID
    unsigned short mode; //权限
    unsigned short seq; //队列号
};
```

消息队列的实现：创建消息队列

原型：int msgget(key_t key,int msgflg);

参数：

- key：命名消息队列的键，一般用 ftok 函数获取
- msgflg：消息队列的访问权限，可以与以下键或操作：
 - IPC_CREAT：不存在则创建，存在则返回已有的 qid

返回值：

- 成功：返回以 key 命名的消息队列的标识符(非零正整数)
- 失败：返回 -1

消息队列的实现：发送消息(将消息添加到消息队列)

原型：int msgsnd(int msgid,const void* msgp,size_t msgsz,int msgflg);

参数：

- msgid：由 msgget 函数返回的消息队列标识符
- msgp：将发往消息队列的消息结构体指针，结构为：

```
struct msgbuf{
    long type; //消息类型，由用户自定义
    消息数据 //发送的消息(长度、类型可以自行指定)，如 char mtext[1024];
};
```
- msgsz：消息长度，是消息结构体中待传递数据的大小(不是整个结构体的大小)
- msgflg：
 - IPC_NOWAIT：消息队列满时返回 -1
 - 0：消息队列满时阻塞

返回值：

- 成功：消息数据的一份副本将被放到消息队列中，并返回 0
- 失败：返回 -1

消息队列的实现：接收消息(从消息队列中获取消息)

Ø 原型: `ssize_t msgrcv(int qid, void *msgp, size_t msgsz, long msgtype, int msgflg);`

Ø 参数:

- ✓ `msgid`、`msgp`、`msgsz`、`msgflg` 的作用同函数 `msgsnd`
- ✓ `msgtype`: 可以实现一种简单的接收优先级。如果 `msgtype` 为 0, 就获取队列中的第一个消息。如果它的值大于零, 将获取具有相同消息类型的第一个信息。如果它小于零, 就获取类型等于或小于 `msgtype` 的绝对值的第一个消息

Ø 返回值:

- ✓ 成功: 返回放到接收缓冲区中的字节数, 消息被复制到由 `msgp` 指向的用户分配的缓冲区中, 然后删除消息队列中的对应消息
- ✓ 失败: 返回 -1

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 37

消息队列的实现：控制消息队列

Ø 原型: `int msgctl(int msgid, int cmd, struct msgid_ds *buf);`

Ø 参数:

- ✓ `msgid`: 由 `msgget` 函数返回的消息队列标识符
- ✓ `cmd`: 将要采取的动作, 它可以取 3 个值之一:
 - IPC_STAT** 用来获取消息队列信息, 并存储在 `buf` 指向的 `msgid_ds` 结构
 - IPC_SET** 用来设置消息队列的属性, 要设置的属性存储在 `buf` 指向的 `msgid_ds` 结构中
 - IPC_RMID** 删除 `msgid` 标识的消息队列
- ✓ `buf`: 指向 `msgid_ds` 权限结构, 它至少包括以下成员:

```
struct msgid_ds
{
    uid_t shm_perm.uid;
    uid_t shm_perm.gid;
    mode_t shm_perm.mode;
    .....
};
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 38

例：接收消息msgreceive.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/msg.h>

struct msg_st
{
    long int    msg_type;
    char        text[BUFSIZ];
};

int main()
{
    int running = 1;
    int msgid = -1;
    struct msg_st data;
    long int msgtype = 0; // 消息类型, 0 表示获取队列中第一个可用的消息,
                          // 注意收发消息类型的对应关系
}
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 39

例：接收消息msgreceive.c(续)

```
//建立消息队列
msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
if(msgid == -1)
{
    fprintf(stderr, "msgget failed with error: %d\n", errno);
    exit(EXIT_FAILURE);
}
//从队列中获取消息, 直到遇到 end 消息为止
while(running)
{
    if(msgrcv(msgid, (void*)&data, BUFSIZ, msgtype, 0) == -1)
    {
        fprintf(stderr, "msgrcv failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }

    printf("You wrote: %s\n", data.text);
    //遇到 end 结束
    if(strcmp(data.text, "end", 3) == 0) // 不区分大小写的字符串比较
        running = 0; // 结束运行
}
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 40

例：接收消息msgreceive.c(续)

```
//删除消息队列
if(msgctl(msgid, IPC_RMID, 0) == -1)
{
    fprintf(stderr, "msgctl(IPC_RMID) failed\n");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
```

```
[root@localhost ~]# ./msgreceive
You wrote: this is a text

You wrote: my name is msgsend.c

You wrote: end

[root@localhost ~]# █
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 41

例：发送信息msgsend.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/msg.h>
#include <errno.h>

#define MAX_TEXT 512
struct msg_st
{
    long int    msg_type;
    char        text[MAX_TEXT];
};

int main()
{
    int running = 1;
    struct msg_st data;
    char buffer[BUFSIZ];
    int msgid = -1;
}
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 42

例：发送信息msgsend.c(续)

```
//建立消息队列
msgqid = msgget((key_t)1234, 0666 | IPC_CREAT);
if(msgqid == -1)
{
    fprintf(stderr, "msgget failed with error: %d\n", errno);
    exit(EXIT_FAILURE);
}

//向消息队列中写消息，直到写入end
while(running)
{
    printf("Enter some text: "); //输入数据
    fgets(buffer, BUFSIZ, stdin);
    data.msg_type = 1; //设置发送的信息类型为1，注意收发消息类型的对应关系

    strcpy(data.text, buffer);
    //向队列发送数据
    if(msgsnd(msgqid, (void*)&data, MAX_TEXT, 0) == -1)
    {
        fprintf(stderr, "msgsnd failed\n");
        exit(EXIT_FAILURE);
    }
}
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 43

例：发送信息msgsend.c(续)

```
//输入 end 结束输入
if(strcmp(buffer, "end", 3) == 0)
    running = 0;
sleep(1);
}
exit(EXIT_SUCCESS);
}
```

```
[root@localhost ~]# ./msgsend
Enter some text:this is a text
Enter some text:my name is msgsend.c
Enter some text:end
[root@localhost ~]#
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 44

消息队列可能存在的隐患

Ø当 64 位应用向 32 位应用发送一消息时，如果它在 8 字节字段中设置的值大于 32 位应用中 4 字节类型字段可表示值，那么 32 位应用在其 mtype 字段中得到的是一个截短了的值，于是也就丢失了信息

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 45

进程通信：信号 (signal)

Ø什么是信号

- ✓信号用于通知接收进程某个事件已经发生
- ✓除了用于进程间通信外，还可以发送信号给进程本身

Ø信号的产生

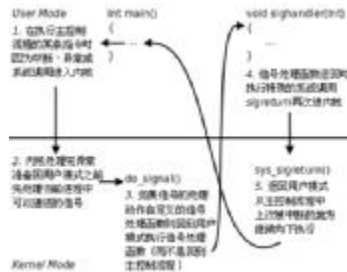
- ✓1) 由硬件产生，如从键盘输入 Ctrl+C 可以终止进程
- ✓2) 由其他进程发送，如 shell下用命令 kill -信号标号 PID 可以向指定进程发送信号
- ✓3) 异常，进程异常时会发送信号

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 46

信号的处理

Ø信号是由操作系统来处理的，说明信号的处理在内核态
Ø信号不一定会立即被处理，此时会储存在信号的信号表中

Ø处理过程：



Ø信号处理的三种方式：

- ✓1) 忽略
- ✓2) 默认处理方式：操作系统设定的默认处理方式
- ✓3) 自定义信号处理方式：可自定义信号处理函数

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 47

信号的处理：signal 函数(自定义信号处理)

Ø头文件：signal.h

Ø原型：void (*signal(int sig, void (*func)(int)))(int);

Ø功能：用于处理指定的信号，主要通过忽略和恢复其默认行为来工作

Ø参数：

- ✓sig：信号值
 - ✓func：信号处理函数指针，参数为信号值
- 注意：信号处理函数的原型必须为void func(int)，或者是下面的特殊值：

u SIG_IGN 忽略信号的处理
u SIG_DFL 恢复信号的默认处理

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 48

例1：自定义信号处理(signal1.c)

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig)
{
    printf("\nOUCH! - I got signal %d\n", sig);
    signal(SIGINT, SIG_DFL); //恢复终端中断信号 SIGINT 的默认行为
}

int main()
{
    // 改变终端中断信号 SIGINT 的默认行为，使之执行 ouch 函数，而不是终止程序
    signal(SIGINT, ouch); // 注册信号处理
    while(1)
    {
        printf("Hello World!\n");
        sleep(1);
    }
    return 0;
}
```

第一次按下终止命令 (Ctrl+c) 时，进程并没有被终止，而是输出“OUCH! - I got signal 2”，因为 SIGINT 的默认行为被 signal 函数改变了，当进程接收到信号 SIGINT 时，它就去调用函数 ouch 去处理。注意 ouch 函数把信号 SIGINT 的处理方式改变成默认的方式，所以当你再按一次 Ctrl+c 时，进程就被终止了

root@localhost ~# gcc -o signal1 signal1.c
root@localhost ~# ./signal1
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
^C

网络安全与网络工程系系东平 jsxbhc@163.com

Linux

root@localhost ~#

例2：自定义信号处理(signal1.c)

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
void my_func(int sign)
{
    if(sign == SIGINT)
        printf("I have get SIGINT\n");
    else if(sign == SIGQUIT)
        printf("I have get SIGQUIT\n");
}

int main()
{
    printf("Waiting for signal SIGINT or SIGQUIT\n");
    signal(SIGINT, my_func); /*注册信号处理函数*/
    signal(SIGQUIT, my_func); /*注册信号处理函数*/
    pause();
    exit(0);
}
```

root 1948 0.0 0.0 3824 416 tty1 S+ 15:37 0:00 ./signal1
root 1965 0.0 0.1 118240 1168 tty2 R+ 15:39 0:00 ps aux
root@localhost ~# kill -s SIGQUIT 1948
root@localhost ~#
root@localhost ~# ./signal1
Waiting for signal SIGINT or SIGQUIT
I have get SIGQUIT
root@localhost ~#

视频：47 进程通信：自定义信号处理

网络安全与网络工程系系东平 jsxbhc@163.com

Linux操作系统

2020年3月2日 4时50分

50

常见信号

- Ø **SIGHUP**：在用户终端结束时发出。通常是在终端的控制进程结束时，通知同一会话期内的各个作业，这时他们与控制终端不再关联。比如，登录 Linux 时，系统会自动分配给登录用户一个控制终端，在这个终端运行的所有程序，包括前台和后台进程组，一般都属于同一个会话。当用户退出时，所有进程组都将收到该信号，这个信号的默认操作是终止进程。此外对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件
- Ø **SIGINT**：程序终止信号。当用户按下 CRTL+C 时通知前台进程组终止进程
- Ø **SIGQUIT**：Ctrl+\ 控制，进程收到该信号退出时会产生 core 文件，类似于程序错误信号
- Ø **SIGILL**：执行了非法指令。通常是因为可执行文件本身出现错误，或者数据段、堆栈溢出时也有可能产生这个信号
- Ø **SIGTRAP**：由断点指令或其他陷阱指令产生，由调试器使用
- Ø **SIGABRT**：调用 abort 函数产生，将会使程序非正常结束
- Ø **SIGBUS**：非法地址，包括内存地址对齐出错。比如访问一个 4 个字节的整数，但其地址不是 4 的倍数。它与 SIGSEGV 的区别在于后者是由于对合法地址的非法访问触发

网络安全与网络工程系系东平 jsxbhc@163.com

Linux操作系统

2020年3月2日 4时50分

51

常见信号(续)

- Ø **SIGFPE**：发生致命的算术运算错误
- Ø **SIGKILL**：用来立即结束程序的运行
- Ø **SIGUSR1**：留给用户使用，用户可自定义
- Ø **SIGSEGV**：访问未分配给用户的内存区。或操作没有权限的区域
- Ø **SIGUSR2**：留给用户使用，用户可自定义
- Ø **SIGPIPE**：管道破裂信号。当对一个读进程已经运行结束的管道执行写操作时产生
- Ø **SIGALRM**：时钟定时信号。由alarm函数设定的时间终止时产生
- Ø **SIGTERM**：程序结束信号。shell使用kill产生该信号，当结束不了该进程，尝试使用SIGKILL信号
- Ø **SIGSTKFLT**：堆栈错误
- Ø **SIGCHLD**：子进程结束，父进程会收到。如果子进程结束时父进程不等待或不处理该信号，子进程会变成僵尸进程
- Ø **SIGCONT**：让一个停止的进程继续执行
- Ø **SIGSTOP**：停止进程执行。暂停执行

网络安全与网络工程系系东平 jsxbhc@163.com

Linux操作系统

2020年3月2日 4时50分

52

常见信号(续)

- Ø **SIGTSTP**：停止运行，可以被忽略Ctrl+z
- Ø **SIGTTIN**：当后台进程需要从终端接收数据时，所有进程会收到该信号，暂停执行
- Ø **SIGTTOU**：与SIGTTIN类似，但在写终端时产生
- Ø **SIGURG**：套接字上出现紧急情况时产生
- Ø **SIGXCPU**：超过CPU时间资源限制时产生的信号
- Ø **SIGXFSZ**：当进程企图扩大文件以至于超过文件大小资源限制时产生
- Ø **SIGVTALRM**：虚拟使用信号。计算的是进程占用CPU调用的时间
- Ø **SIGPROF**：包括进程使用CPU的时间以及系统调用的时间
- Ø **SIGWINCH**：窗口大小改变时
- Ø **SIGIO**：文件描述符准备就绪，表示可以进行输入输出操作
- Ø **SIGPWR**：电源失效信号
- Ø **SIGSYS**：非法的系统调用

网络安全与网络工程系系东平 jsxbhc@163.com

Linux操作系统

2020年3月2日 4时50分

53

信号处理：sigaction 函数

- Ø **原型**：int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
- Ø **功能**：设置与信号 sig 关联的动作
- Ø **参数**：
 - ✓ sig：信号值
 - ✓ act：指定信号的动作
 - ✓ oact：保存原信号的动作
- Ø **注意**：sigaction 函数有阻塞的功能
 - ✓ 默认情况下，在信号处理函数未完成之前，发生的新的 SIGINT 信号将被阻塞，同时对后续来的 SIGINT 信号进行排队合并处理

网络安全与网络工程系系东平 jsxbhc@163.com

Linux操作系统

2020年3月2日 4时50分

54

信号处理：sigaction 函数(续)

Ø sigaction 结构体：

```
struct sigaction
{
    void (*)(int) sa_handler; //处理函数指针，相当于 signal 函数的 func 参数
    sigset_t sa_mask; //被屏蔽的信号，可以消除信号间的竞态
    int sa_flags; //处理函数执行完后，信号处理方式修改。如 SA_RESETHAND，
    //将信号处理方式置为 SIG_DFL
```

- ✓ 信号屏蔽集 sa_mask 可以通过函数 sigemptyset / sigaddset 等来清空和增加需要屏蔽的信号
- ✓ sa_flags
 - F 0：默认行为
 - F SA_NODEFER：不进行当前处理信号到阻塞
 - F SA_RESETHAND：处理完当前信号后，将信号处理函数设置为 SIG_DFL 行为

信号的阻塞

Ø 阻塞是阻止进程收到该信号，此时信号处于未决状态，放入进程的未决信号表中，当解除对该信号的阻塞时，未决信号会被进程接收

Ø 1) 阻塞信号

✓ 原型：int sigprocmask(int how, const sigset_t *set, sigset_t *oset);

✓ how：设置 block 阻塞表的方式

F a.SIG_BLOCK：将信号集添加到 block 表中

F b.SIG_UNBLOCK：将信号集从 block 表中删除

F c.SIG_SETMASK：将信号集设置为 block 表

✓ set：要设置的集合

✓ oset：设置前保存之前 block 表信息

Ø 2) 获取未决信号

✓ 原型：int sigpending(sigset_t *set);

✓ set：存储获得的当前进程的 pending 未决表中的信号集

例：sigaction 函数(unmask.c)

```
#include <stdio.h>
#include <signal.h>
void ouch(int sig)
{
    printf("\nOUCH! - I got signal %d\n", sig);
}
```

int main()

```
{
    struct sigaction act;
    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask); //创建空的信号屏蔽字，即不屏蔽任何信息
    act.sa_flags = SA_RESETHAND; //使 sigaction 函数重置为默认行为
    sigaction(SIGINT, &act, 0);
    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
    return 0;
}
```

视频：48 进程通信：sigaction(不屏蔽)

```
root@localhost ~# gcc -o unmask unmask.c
root@localhost ~# ./unmask
Hello World!
Hello World!
Hello World!
Hello World!
OUCH! - I got signal 2
Hello World!
Hello World!
root@localhost ~#
```

例：sigaction 函数(mask.c)

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static void sig_quit(int signo)
{
    printf("catch SIGQUIT\n");
    signal(SIGQUIT, SIG_DFL);
}
```

int main (void)

```
{
    sigset_t new, old, pend;
    signal(SIGQUIT, sig_quit);
    sigemptyset(&new);
    sigaddset(&new, SIGQUIT);
    sigprocmask(SIG_BLOCK, &new, &old);
    sleep(5);
    printf("SIGQUIT unblocked\n");
    sigprocmask(SIG_SETMASK, &old, NULL);
    sleep(20);
    return 1;
}
```

```
root@localhost ~# gcc -o mask mask.c
root@localhost ~# ./mask
SIGQUIT unblocked
catch SIGQUIT
root@localhost ~#
```

视频：49 进程通信：sigaction(屏蔽)

信号的发送：kill 函数

Ø 原型：int kill(pid_t pid, int sig);

Ø 功能：发送信号给进程或进程组，可以是本身或其它进程

Ø 注意：不能误以为 kill() 就是 kill

Ø 参数：

✓ pid：

F >0：发送给进程 ID

F 0：发送给所有和当前进程在同一个进程组的进程

F -1：发送给所有的进程表中的进程(进程号最大的除外)

F <-1：发送给进程组号为 -PID 的每一个进程

✓ sig：信号值

Ø 返回值：

✓ 成功：返回 0

✓ 失败：返回 -1，失败通常有三大原因

F 1) 给定的信号无效(errno = EINVAL)

F 2) 发送权限不够(errno = EPERM)

F 3) 目标进程不存在(errno = ESRCH)

信号的发送：raise 函数

Ø 原型：int raise(int sig);

Ø 参数：

✓ sig：信号值

Ø 注意：只能向进程自身发信号

Ø 返回值：

✓ 成功：0

✓ 失败：-1

信号的发送：abort 函数

Ø 功能：发送 SIGABRT 信号，让进程异常终止，发生转储 (core)
Ø 原型：void abort(void);

信号的发送：pause 函数

Ø 原型：int pause(void);
Ø 返回值：
✓ 成功：0
✓ 失败：-1，同时把 errno 设置为 EINTR
Ø 说明：
✓ pause() 函数用于将调用进程挂起直至捕捉到信号为止
✓ 这个函数通常可以用于判断信号是否已到

信号的发送：alarm 函数(闹钟函数)

Ø 功能：发送 SIGALRM 闹钟信号
Ø 原型：int alarm(unsigned int seconds);
✓ 参数：
F seconds：系统经过 seconds 秒后向进程发送 SIGALRM 信号
✓ 返回值：
F 成功：如果调用 alarm() 前，进程中已经设置了闹钟时间，则返回上一个闹钟的剩余时间，否则返回
F 失败：-1
Ø 说明：可以在进程中设置一个定时器，当定时器指定的时间到时，它就向进程发送 SIGALARM 信号
Ø 注意：一个进程只能有一个闹钟时间，如果在调用 alarm() 之前已设置过闹钟时间，则任何以前的闹钟时间都被新值所代替

fork、sleep 和 signal 函数，用一个例子来说明 kill 函数

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

static int alarm_fired = 0;

void ouch(int sig)
{
    alarm_fired = 1;
    signal(SIGALRM, ouch);
    while(!alarm_fired)
    {
        printf("Hello World!\n");
        sleep(1);
    }
}

int main()
{
    pid_t pid;
    pid = fork();
    switch(pid)
    {
        case -1:
            perror("fork failed\n"); // 先显示字符串
            exit(0);
        case 0:
            printf("I got a signal %d\n", SIGALRM);
            exit(0);
    }
}

// 再显示 errno 对应的字符串
case 0: // 子进程
    sleep(5);
    /* 向父进程发送信号 */
    kill(getppid(), SIGALRM);
    exit(0);
}

// 设置处理函数
signal(SIGALRM, ouch);
while(!alarm_fired)
{
    printf("Hello World!\n");
    sleep(1);
}

if(alarm_fired)
    printf("I got a signal %d\n", SIGALRM);
exit(0);

// 运行视频：50 进程通信：testkill.c
```

kill 函数和 raise 函数(testkillraise.c)

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

int main()
{
    pid_t pid;
    int ret;
    if( (pid=fork())<0 ) // 创建一个子进程，出错处理
    {
        printf("Fork error\n");
        exit(1);
    }
}
```

kill 函数和 raise 函数

```
else if(pid==0) // 子进程
{
    // 在子进程中使用 raise() 函数发出 SIGSTOP 信号，使子进程暂停
    printf("I am child progress(pid:%d).I am waiting for any signal\n",getpid());
    raise(SIGSTOP);
    printf("I am child progress(pid:%d). I am killed by progress:%d\n",
        getpid(), getppid());
    exit(0);
}
else // 父进程
{
    /* sleep(2); */ // 先让父进程休眠，让子进程执行，把这句去掉再运行试试
    if(waitpid(pid,NULL,WNOHANG)==0) // 在父进程中收集子进程发出的信号，
        // 并调用 kill() 函数进程相应的操作
        if((ret=kill(pid, SIGKILL))==0) // 若 pid 指向的子进程没有退出，则返回0，
            // 且父进程不阻塞，继续执行下面的语句
            printf("I am parent progress(pid:%d).I kill %d\n", getpid(), pid);
        waitpid(pid, NULL, 0); // 等待子进程退出，否则就一直阻塞
        exit(0);
}

// 运行视频：51 进程通信：testkillraise.c
// 取消 sleep(2)后的运行视频：52 进程通信：testkillraise.c(修改)
```

信号处理函数的安全问题

Ø如果信号处理过程中被中断，再次调用，然后返回到第一次调用时，要保证操作的正确性，这就要求信号处理函数必须是可重入的。可重入函数如下：

access	alarm	cfgetspeed	cfgetospeed	cfsetispeed	sfsetospeed
chdir	chmod	chown	close	create	dup2
dup	execl	execv	_exit	fcntl	fork
fstat	getpid	geteuid	getgid	getgroups	getpgrp
getpid	getppid	getuid	kill	link	lseek
mkdir	mkfifo	open	patchconf	pause	pipe
read	rename	rmdir	setgid	setpgid	setsid
setuid	sigaction	sigaddset	sigdelset	sigemptyset	sigfillset
sigismember	signal	sigpending	sigprocmask	sigsuspend	sleep
stat	sysconf	tcdrain	tcflow	tcflush	tcgetattr
tcgetpgrp	tcsendbreak	tcsetattr	tcsetpgrp	time	times
umask	uname	unlink	utime	wait	waitpid
write					

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 67

例：用信号的知识实现司机售票员问题(不讲)

- Ø1) 售票员捕捉 SIGINT (代表开车)信号，向司机发送 SIGUSR1 信号，司机打印("let's gogogog")
- Ø2) 售票员捕捉 SIGQUIT (代表停车)信号，向司机发送 SIGUSR2 信号，司机打印("stop the bus")
- Ø3) 司机捕捉 SIGTSTP (代表车到终点站) 信号，向售票员发送 SIGUSR1 信号，售票员打印("please get off the bus"), 然后售票员下车
- Ø4) 司机等待售票员下车，之后司机再下车

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 68

例：用信号的知识实现司机售票员问题代码(不讲)

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

static pid_t slr_pid;

void drv_handler(int signo) // 司机
{
    if(signo == SIGUSR1)    printf("driver: let's gogogo\n");
    else if(signo == SIGUSR2) printf("driver: stop the bus\n");
    else if(signo == SIGTSTP) kill(slr_pid, SIGUSR1);
    else if(signo == SIGCHLD)
    {
        wait(NULL);
        printf("main exit\n");
        exit(0);
    }
}
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 69

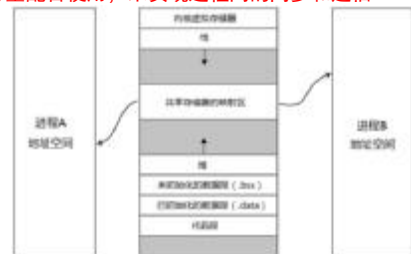
```
void slr_handler(int signo) // 售票员
{
    if(signo == SIGINT)    kill(getppid(), SIGUSR1);
    else if(signo == SIGQUIT) kill(getppid(), SIGUSR2);
    else if(signo == SIGUSR1)
    {
        printf("seller: please get off the car\n");
        printf("child process exit\n");
        exit(0);
    }
}
```

```
int main()
{
    slr_pid = fork();
    if(slr_pid < 0) perror("fork error:"); // 先显示字符串，再显示errno的错误字串
    else if(slr_pid == 0) {
        signal(SIGINT, slr_handler); // Ctrl+c
        signal(SIGQUIT, slr_handler); // Ctrl+\
        signal(SIGUSR1, slr_handler); // 用户自定义
        signal(SIGUSR2, slr_handler); // 用户自定义，忽略信号的处理程序
        signal(SIGTSTP, SIG_IGN); // Ctrl+z, 忽略信号的处理程序
    }
    else {
        signal(SIGINT, SIG_IGN); // 忽略信号的处理程序
        signal(SIGQUIT, SIG_IGN); // 忽略信号的处理程序
        signal(SIGUSR1, drv_handler); // 用户自定义
        signal(SIGUSR2, drv_handler); // 用户自定义
        signal(SIGTSTP, drv_handler); // 用户自定义
        signal(SIGCHLD, drv_handler); // 子进程结束
    }
    while(1) pause();
    exit(0);
}
```

运行视频：53进程通信：driverseller.c

进程通信：共享内存(shared memory)

- Ø共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问
- Ø共享内存是最快的 IPC 方式，往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信



网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 72

创建/获取共享内存：shmget 函数

Ø 头文件： sys/ipc.h
sys/shm.h

Ø 原型： int shmget(key_t key, size_t size, int shmflg);

Ø 参数：

- ✓ key：共享内存段的名字，通常用 ftok() 函数获取
- ✓ size：以字节为单位的共享内存大小。内核以页为单位分配内存，但最后一页的剩余部分内存不可用
- ✓ shmflg：九个比特的权限标志(其作用与文件 mode 模式标志相同)，并与 IPC_CREAT 或时创建共享内存段

Ø 返回值：

- ✓ 成功：非负整数，即该共享内存段的标识码
- ✓ 失败：-1

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 73

将共享内存段连接到进程地址空间：shmat 函数(挂接)

Ø 原型： void *shmat(int shmid, const void *shmaddr, int shmflg);

Ø 说明：共享内存段刚创建时不能被任何进程访问，必须将其连接到一个进程的地址空间才能使用

Ø 参数：

- ✓ shmid：由 shmget 返回的共享内存标识
- ✓ shmaddr：指定共享内存连接到当前进程中的地址位置
 - F shmaddr = NULL：核心自动选择一个地址
 - F shmaddr <> NULL 且 shmflg = SHM_RND：以 shmaddr 为连接地址
 - F shmaddr <> NULL 且 shmflg = SHM_RND：连接的地址会自动向下调整为 SHMLBA 的整数倍，公式：shmaddr - (shmaddr % SHMLBA)
- ✓ shmflg：它有两个可能取值，用来控制共享内存连接的地址
 - F SHM_RND：以 shmaddr 为连接地址
 - F SHM_RDONLY：表示连接操作用来只读共享内存

Ø 返回值：

- ✓ 成功：指向共享内存第一个节的指针
- ✓ 失败：-1

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 74

将共享内存段与当前进程脱离：shmdt 函数(去关联)

Ø 功能：将共享内存从当前进程中分离

Ø 原型： int shmdt(const void *shmaddr);

Ø 参数：

- ✓ shmaddr：shmat 所返回的指针

Ø 返回值：

- ✓ 成功：0
- ✓ 失败：-1

Ø 注意：

- ✓ 共享内存分离并未删除它，只是使得该共享内存对当前进程不再可用

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 75

共享内存控制：shmctl 函数

Ø 原型： int shmctl(int shmid, int cmd, struct shmid_ds *buf);

Ø 参数：

- ✓ shmid：由 shmget 返回的共享内存标识码
- ✓ cmd：将要采取的动作，有三个可取值
 - IPC_STAT 把 shmid_ds 结构中的数据设置为共享内存的当前关联值
 - IPC_SET 如果有足够的权限，把共享内存的当前值设置为 shmid_ds 数据结构中给出的值
 - IPC_RMID 删除共享内存段
- ✓ buf：用于保存共享内存模式状态和访问权限，至少包含以下成员

```
struct shmid_ds {
    uid_t      shm_perm.uid;
    uid_t      shm_perm.gid;
    mode_t     shm_perm.mode;
}
```

Ø 返回值：成功返回0，否则返回-1

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 76

例：写数据进程(writeshm.c)

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

int main()
{
    key_t key = ftok("./", 66); // 生成一个key
    // 创建共享内存，返回一个id
    int shmid = shmget(key, 8, IPC_CREAT|0666|IPC_EXCL);
    if(shmid==-1)
    {
        perror("shmget failed");
        exit(1);
    }
}
```

网络安全与网络工程系系东平 jsxbhc@163.com Linux操作系统 2020年3月2日 4时50分 77

```
// 映射共享内存，得到虚拟地址
void *p = shmat(shmid, 0, 0);
if( p!=(void*)-1 )
{
    perror("shmat failed");
    exit(2);
}
```

```
// 写共享内存
int *pp = p;
*pp = 0x12345678;
*(pp + 1) = 0xffffffff;
```

```
// 解除映射
if( shmdt(p) == -1 )
{
    perror("shmdt failed");
    exit(3);
}
printf("shared-memory released successful, click return to destroy it\n");
getchar();

// 销毁共享内存
if( shmctl(shmid, IPC_RMID, NULL) == -1 )
{
    perror("shmctl failed");
    exit(4);
}

return 0;
}
```

例：读数据进程(readshm.c)

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

int main()
{
    // 生成一个key
    key_t key = ftok("./", 66);

    // 获取共享内存，返回一个id
    int shmid = shmget(key, 0, 0);
    if( shmid == -1 )
    {
        perror("shmget failed");
        exit(1);
    }
}
```

网络安全与网络工程系杨家平 jaxhbc@163.com Linux操作系统 2020年3月2日4时50分 80

```
// 映射共享内存，得到虚拟地址
void *p = shmat(shmid, 0, 0);
if( p == (void*)-1 ) {
    perror("shmat failed");
    exit(2);
}

// 读共享内存
int x = *(int *)p;
int y = *((int *)p + 1);
printf("Read from shared memory:0x%x and 0x%x \n", x, y);

// 解除映射
if( shmdt(p) == -1 ) {
    perror("shmdt failed");
    exit(3);
}

return 0;
}
```

读写共享内存运行视频：54进程通信：writeshm.c和readshm.c

```
root@localhost ~# ./writeshm
shared-memory released successful,click return to destroy it
root@localhost ~# _
root@localhost ~# ./readshm
Read from shared memory:0x12345678 and 0xffffffff
root@localhost ~# _
```

进程通信：套接字 (socket)

Ø在网络编程中讲述

网络安全与网络工程系杨家平 jaxhbc@163.com Linux操作系统 2020年3月2日4时50分 82